Informe de Calidad

Adrián Edreira Gantes

Índice

1. Introducción

Apache Camel es un framework de integración de código abierto que permite definir reglas de enrutamiento en variedad de lenguajes, lo que permite disfrutar de la autocompletación inteligente de dichas reglas en nuestros IDE.

Es una biblioteca pequeña con pocas dependencias, lo que facilita la integración en aplicaciones Java. Además, permite trabajar con la misma API sin importar el tipo de transporte utilizado, por lo que solo necesitas aprenderla una vez para interactuar con todos los componentes disponibles de forma predeterminada. También cuenta con integración fluida con el framework Spring y proporciona herramientas para realizar pruebas unitarias entre las rutas.

Algunos ejemplos de proyectos Apache que se aprovechan de Apache Camel son: Apache ServiceMix, Apache ActiveMQ o Apache MINA.

2. Análisis

Este análisis se realiza con el objetivo de evaluar la situación actual del software, encontrar fallos relevantes y proponer mejoras. Será enfocado en los ejes de calidad de producto definidos en la ISO 25010, aunque algunos de estos campos no sean abordados con la mayor precisión por Sonar, como pueden ser la portabilidad, el rendimiento o la adecuación funcional.

Usabilidad

Como podemos observar en el repositorio de github, disponemos de un readme completo, con una serie de enlaces para ver la lista de componentes soportados, ejemplos, guías de inicio e incluso su propia página de soporte.

En cuanto a la protección de errores, en la pestaña issues, podemos ver que hay algunos bugs relacionados con excepciones y con el control de nulos pendientes por resolver, en ellos podemos ver sus etiquetas, criticidad, el tiempo que nos puede llevar resolverlo o en que nos afecta (Consistencia, adaptabilidad…). Por último, al tratarse de un software de backend, no contiene ningún tipo de práctica de accesibilidad ni estética.

Fiabilidad

La nota que nos muestra SonarQube acerca de la fiabilidad del proyecto actual es una E, con un total de 785 bugs y un tiempo necesario para la solución de ellos de 21 días. La mayoría de estos bugs se encuentran en las carpetas de “components” y “core”, ambas con una calificación E. El archivo “DefaultCamelContextExtension.java” es él que más tiempo llevaría solucionar, con 1 día y 2 horas.

En la pestaña Issues, podemos ver la severidad de estos bugs, 90 de ellos tienen una severidad alta, 348 una severidad media y el resto una severidad baja. Vemos que del total, 9 de ellos son de consistencia, considerados leves; 22 de adaptabilidad, considerados medios; y el resto de intencionalidad, con diferentes severidades.

Inspeccionando más a fondo vemos que los errores de consistencia se deben a la falta de excepciones, llevaría 5 minutos solucionar cada uno y algunos llevan sin solucionarse más de 10 años. Los bugs de adaptabilidad son causados por métodos no sincronizados que podrían dar errores de concurrencia. Por último, entre los problemas de intencionalidad, vemos que hay varios problemas causados por la ausencia de cierre de conexiones, otros causados por comparaciones en los tests, algunos en getters y setters y uno que he podido encontrar de un loop sin condición de parada.

Si profundizamos más en cada uno de los problemas de mayor severidad, podemos ver que no son tan críticos como parecen. Los problemas causados por el cierre de conexiones podrían ser de vital importancia, sin embargo, se usa un pool de conexiones, por lo que tiene sentido que no se cierren las conexiones cuando una vez utilizadas se devuelven a dicho pool. Las comparaciones en los test indican que son valores de diferente tipo al comparar un “int” con un “Integer”, los getters y setters son falsos positivos y el bucle sin condición de parada es un socket abierto para la escucha de peticiones web.

Tras esto, podemos valorar que, aunque Sonar nos muestra una valoración E, la realidad es diferente y que la nota del proyecto es mayor a la presentada en cuanto a fiabilidad se refiere.

Compatibilidad

Como podemos leer en su página web y en su readme, Apache Camel tiene pocas dependencias con otras librerías y la capacidad de integración con otros sistemas. En el repositorio de github podemos encontrar una carpeta llamada “test-infra” que dentro contiene gran variedad de carpetas para testear la compatibilidad con diferentes infraestructuras y sistemas. Además, como resulta obvio, al tratarse de un software diseñado para la comunicación y enrutamiento de mensajes, tiene capacidad para compartir entornos con otros sistemas sin causar conflictos.

Aunque es necesario hacer pruebas en entornos reales para valorar su interoperabilidad, en su página web podemos encontrar una pestaña llamada historias de usuarios que nos muestra compañías, productos y proyectos que usan esta tecnología y una descripción de su experiencia, lo que nos sirve para comprobar dicha interoperabilidad en diferentes entornos.

Seguridad

La nota obtenida en Sonar en cuanto a seguridad es una E, encontrando un total de 9 vulnerabilidades y 107 security hotspots, el archivo causante de esta valoración es AbstractExchange.java, a causa de una función que puede ejecutar lo que se le pasa por parámetro para convertirlo a un tipo específico sin hacer comprobaciones, lo que podría ser materializado en una amenaza.

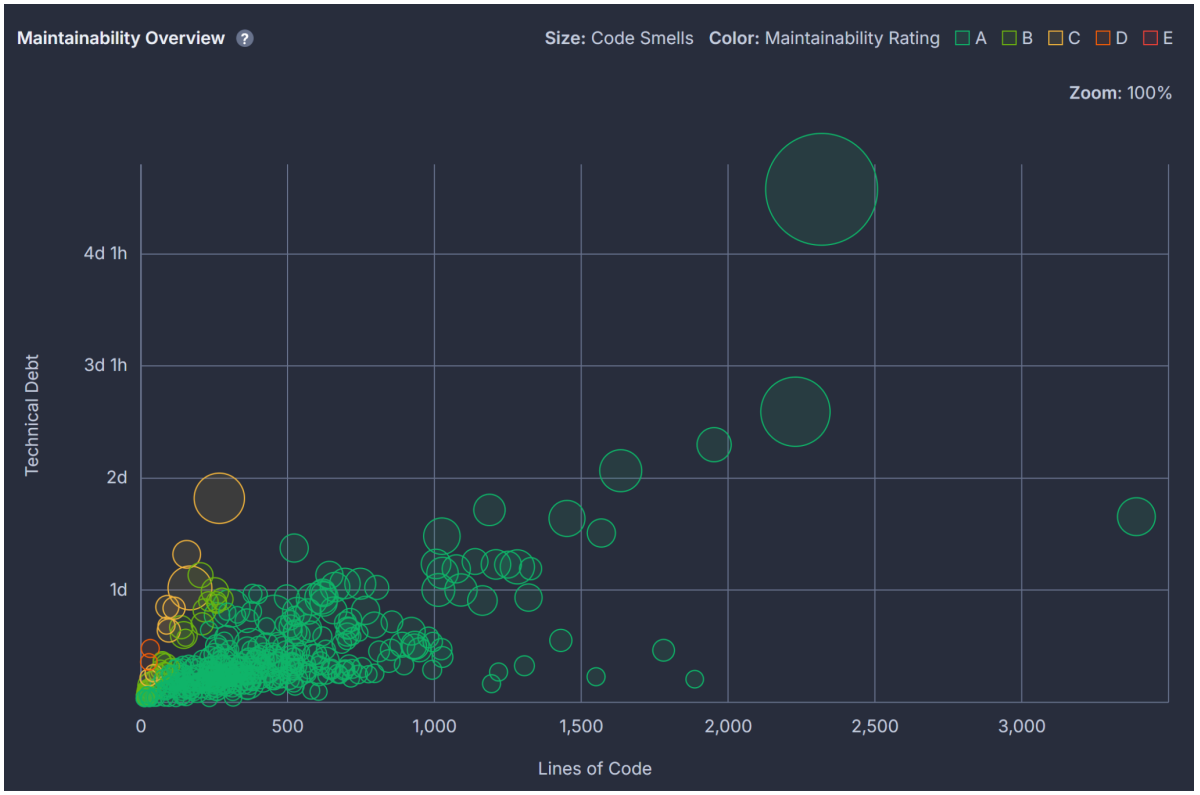
Entre las vulnerabilidades podemos observar que todas representan problemas de injection, partes en las que usuarios podrían inyectar código malicioso para acceder a datos que no deberían, por ejemplo. Una de ellas es bloqueante, la nombrada con anterioridad, ya que el usuario podría introducir una petición HTTP maliciosa o una sentencia SQL y al invocarse el método por el que es usado podría propagar el contenido malicioso o obtener información que no debería. El resto de ellas son de severidad media o baja. El tiempo necesario para remediar estos problemas de seguridad es un total de 4 horas y media.

Por otra parte, los security hotspots encontramos diferentes motivos por los que pueden ser problemas de seguridad, como pueden ser configuraciones de loggers, configuraciones de permisos, inyecciones de SQL o posibles denegaciones de servicio. Estos problemas deben ser revisados manualmente y ser clasificados ya sea como falsa alarma o como vulnerabilidad y asignarle su correspondiente criticidad. Por ejemplo, al ser una arquitectura por capas, hay que asegurarse de que a los DAOs no les lleguen inyecciones de SQL, ya que estos las ejecutan sin hacer comprobaciones; y esta es una de esas brechas de seguridad que Sonar nos advierte.

Mantenibilidad

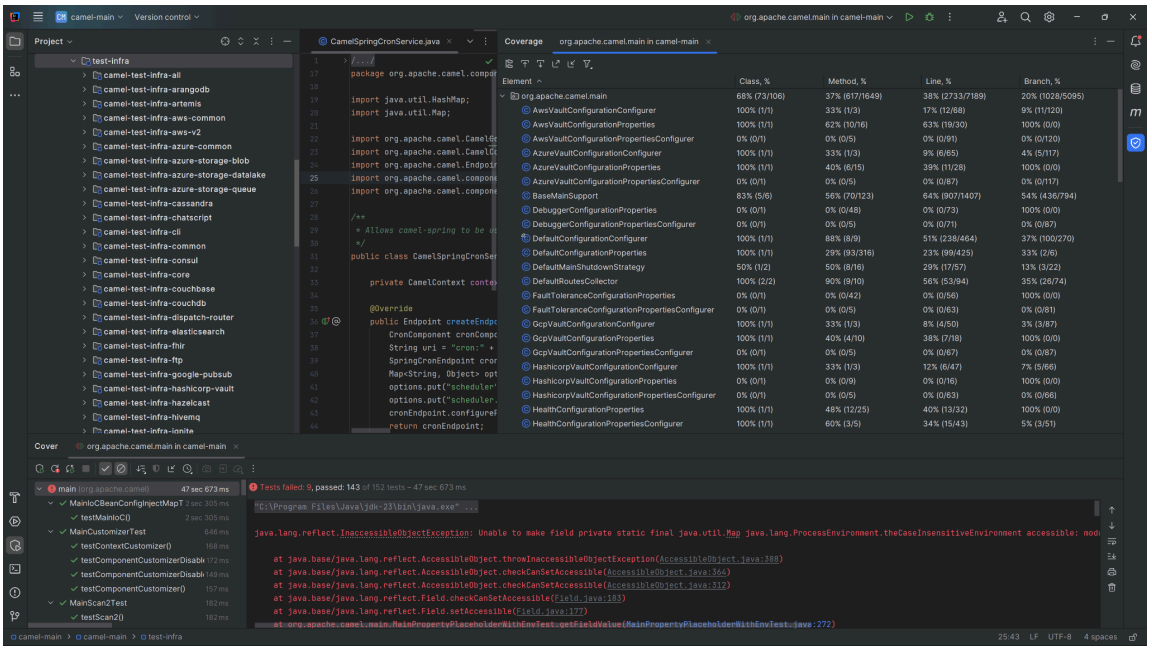
La mantenibilidad de este proyecto está bien conseguida como hemos ido viendo a lo largo de los demás campos de la ISO 25010, pues el código está dividido de forma lógica, por capas, para adaptarse a diferentes arquitecturas y facilitar cambios a lo largo del ciclo de vida del software, por lo que Sonar le designa una nota de A en este apartado. Si miramos en la pestaña Issues vemos que hay 23 mil problemas de mantenibilidad, 283 de ellos bloqueantes. Sin embargo, podemos ver que la mayoría son problemas de comparaciones en los test, por lo que no debería bloquear el correcto funcionamiento del software.

Si observamos las estadísticas, tiene un ratio de deuda técnica de tan solo el 1.4%, aunque en días son 667, y un total de 23.006 códigos malolientes. Si nos fijamos en la gráfica podemos ver que MXParser.java es el archivo con más deuda técnica con un total de 4 días y 7 horas, 224 líneas de código maloliente y una calificación A. Sin embargo, JmsObjectFactory.java y RouteController.java tienen una calificación D y entre los dos suman un total de 7 horas de deuda técnica y 21 líneas de código maloliente en un total de 59 líneas, por lo que sería interesante revisarlos, y de ser el caso, priorizarlos.



Si nos fijamos en la complejidad cognitiva, vemos que hay un total de 107.311 líneas de código para loops y ifs, entre otras. Esto puede hacer más complejo la lectura del software y con ello el mantenimiento del mismo, además de afectar a la usabilidad para cualquier desarrollador que quiera modificarlo. En cuanto a la complejidad ciclomática, que tiene un valor de 147.903 líneas, complica la realización de pruebas, ya que se necesitaría esa cantidad de pruebas para poder testear todas las características del código y que todas funcionen a la perfección.

En relación con las pruebas, una métrica que Sonar no nos proporciona para este proyecto es la cobertura de los tests (Coverage). No obstante, disponemos del código a través de github el cual pude descargar y conseguí ejecutar sus tests de forma local. Tras hacer esto, como se ve en la siguiente imagen, pude obtener una cobertura total del 68% para las clases, un 37% para los métodos, un 38% para las líneas y un 20% para las ramas condicionales. De esto podemos deducir que la cobertura general de los tests es bastante baja, salvo las clases que podríamos considerarla aceptable. Además, vemos que 9 de los 153 test que se han realizado han fallado, por lo que podemos decir que han sido exitosos ya que han encontrado algún error.



Cabe destacar que, aunque tiene cerca de un 7% de duplicación de código, vemos que la gran mayoría de código duplicado se encuentra en la carpeta de componentes. Esto se debe a necesidades de integración con diversas tecnologías y de compatibilidad con diferentes versiones y protocolos; lo que conlleva a usar estructuras repetitivas o adaptaciones del código existente para cada tecnología, permitiendo tener mejor estructurado el código a la hora de modificar un componente en específico. No obstante, corregir un error podría hacer necesario corregirlo en diferentes partes del código.

Por último, el tamaño del código supera los 750 KLOC, donde la gran mayoría de LOC se encuentran en las carpetas components, core y dsl. Es necesario controlar este valor, pues un crecimiento excesivo y sin control podría aumentar el número de duplicaciones y la complejidad, con la consecuencia de afectar a la mantenibilidad negativamente.

Portabilidad

Para poder valorar la portabilidad es necesario hacerlo en entornos reales mediante pruebas de instalación. Sin embargo, usando la misma página que usamos para medir la compatibilidad, podemos ver la variedad de sistemas a los que es portable el software actual y su facilidad de instalación. Además, al hacer un uso mínimo de dependencias con otras librerías y usar el lenguaje Java, permite ejecutarse en cualquier máquina con una JVM adecuada.

Eficiencia

Para medir la eficiencia es necesario realizar pruebas dinámicas y benchmarking, que permiten medir el rendimiento de software para mejorarlo, optimizarlo y compararlo. Sonar no permite evaluar este campo con facilidad, pero si quisiéramos cuantificar la eficiencia deberíamos establecer un objetivo, seleccionar una unidad de medida (por ejemplo, comunicaciones realizadas en un minuto), medir la ejecución de procesar un conjunto de datos y analizar los resultados obtenidos.

Adecuación

Al igual que con la eficiencia y la portabilidad, la adecuación es complicada de valorar. A través de Sonar resulta imposible conseguirlo, ya que no puede determinar si cumple con los requisitos funcionales del usuario final o del cliente. La manera más acertada de puntuar esta característica sería mediante tests de aceptación, es decir, pruebas realizadas con el cliente para ver si los requisitos funcionales que se esperaban del software se cumplen completamente, correctamente y pertinentemente.

Evolución en el tiempo

Por último, vemos que el proyecto ha seguido una mejora continua desde que se hace uso de Sonar, pues aunque el proyecto se inició en 2008, Sonar solo mantiene un registro desde 2022. Desde el inicio se redujo la cantidad de problemas en el código, pasando de 44 mil a menos de 18 mil, estando actualmente por debajo de los 25 mil; las duplicaciones se han mantenido por debajo del 7%, esto debido a los motivos explicados con anterioridad; las vulnerabilidades también se redujeron, pasando de más de 60 a menos de 10 en la actualidad; y el ratio de deuda técnica se mantiene por debajo del 2%, y se ha ido reduciendo paulatinamente desde el inicio. Con esto, podemos deducir que el proyecto está teniendo una progresión adecuada desde que se ha comenzado a usar Sonar.

3. Plan de acción

Para solucionar los problemas detectados en el análisis vamos a focalizar los esfuerzos en los siguientes aspectos por orden de importancia y prioridad:

1. Seguridad

   Es de vital importancia resolver los problemas de inyecciones de SQL, es uno de los problemas que más afecta al software según las OWASP Top 10 y es impermisible que se puedan ejecutar sentencias SQL por usuarios que no deben poder leer, modificar ni borrar información a la que no tienen acceso. Además, es interesante revisar los security hotspots y corregirlos de ser el caso, pues Sonar puede detectarlo pero no definirlo como vulnerabilidad y podría ser un riesgo grave, necesario de corregir cuanto antes.

2. Mantenibilidad

   Lo primordial en relación a la mantenibilidad son los tests. Como vimos, hay 9 tests que no fueron superados, por lo que es necesario corregir los fallos cuanto antes, ya que podrían acarrear problemas mayores como pérdidas de información o denegaciones de servicio. Además, son escasos y no permiten comprobar de forma correcta todo lo que se ha codificado, más del 60% de métodos y líneas no han sido probadas, y el 80% de sentencias condicionales no se han ejecutado, lo que es inaceptable y debería ser solucionado cuanto antes.

   Por otra parte, sería interesante revisar las partes de código maloliente nombradas con anterioridad, pues pueden acarrear problemas mayores en el futuro llegando a hacer el software inmantenible. Aunque las estadísticas están a nuestro favor con un 1.4% de ratio de deuda técnica, si dejamos que estos valores se incrementen significativamente llegará un punto que harán que el software resulte inutilizable.

3. Fiabilidad

   En cuanto a la fiabilidad, sería interesante comprobar todas aquellas issues calificadas con una severidad alta, pues podrían acarrear en problemas mayores. Gran parte de estos problemas o carencias serían solucionados por unos tests de calidad que saquen a relucir todos estos problemas y así poderlos corregir. Debido a la cantidad de errores reportados, encontrar alguno que pueda ser realmente crítico se puede convertir en una ardua tarea si no realizamos unos tests adecuados.

4. Otros aspectos

   Sería interesante incluir en la página web o en el repositorio de github resultados de pruebas de portabilidad, compatibilidad y eficiencia. De esta forma se podría demostrar y documentar estos aspectos de la ISO 25010, ayudando así a mostrar a posibles clientes que nuestro software no tendrá problemas para funcionar en su sistema.

4. Quality Gate

Los siguientes criterios son los que deberán bloquear la puesta en producción del software:

- Cobertura de los tests unitarios inferior al 80%: No podemos afirmar que nuestro software se comporta como esperamos si no tenemos tests que lo demuestren, por eso es necesario mejorar los tests para evitar problemas en la fase de producción.

- Vulnerabilidades relacionadas con la integridad y privacidad de la información: Al tratarse de un software para enrutamiento de información es importante que esa información esté protegida y no haya accesos no permitidos a dicha información.

- Pruebas no exitosas en ciertos sistemas: La incompatibilidad con ciertos sistemas deberá bloquear la puesta en producción en dichos sistemas, pues no se puede asegurar el funcionamiento correcto si se producen errores en las pruebas de compatibilidad y/o portabilidad.

- Media o baja fiabilidad: Si nuestro software no es fiable a la hora de enrutar la información y/o comete errores que bloqueen su ejecución, como pueden ser excepciones no manejadas correctamente, deberán ser arreglados antes de poner en producción. Estos errores pueden causar la pérdida de usuarios debido a la mala calidad del software.

- Baja eficiencia: Es primordial que la comunicación de mensajes sea lo más rápida y eficaz posible. En caso de no cumplirse, habrá que mejorar este aspecto, pues un bajo rendimiento puede ser causa de abandono del software.

Como ya expliqué con anterioridad, las pruebas relacionadas con la eficiencia, compatibilidad y portabilidad no son evaluables con Sonar, pero si que son de alta importancia para este software en concreto. Por este motivo están incluidos en el Quality Gate y deberán ser ejecutadas y aprobadas las pruebas relacionadas con estos campos mediante pruebas en diferentes máquinas.

5. Bibliografía

Página Oficial Apache Camel: https://camel.apache.org/

Historias de Usuarios Apache Camel: https://camel.apache.org/community/user-stories/

Sonar Qube for Apache Camel: https://sonarcloud.io/project/overview?id=apache_camel

Github for Apache Camel: https://github.com/apache/camel

ISO 25010: https://iso25000.com/index.php/normas-iso-25000/iso-25010

Chat GPT: https://chatgpt.com/