



Máster en Cloud Apps: Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2020/2021

Trabajo de Fin de Máster

Reactor en Spring vs Mutiny en Quarkus

[Repositorio Github](#)

Autores: Izan Santana Alonso y Diego Fernandez Aceves

Tutor: Micael Gallego Carrillo



1. Introducción y objetivos	3
¿Por qué está comparativa?	3
¿Por qué Quarkus?	3
2. Introducción a Quarkus	4
¿Cómo ha sido el paso a Quarkus?	4
Comunidad y documentación sobre Quarkus	4
Quarkus con Mutiny	5
Comparativa con Spring Webflux	7
Curva de aprendizaje	7
3. Comparativa Quarkus vs Spring	8
Desarrollo de aplicaciones con diferentes tecnologías	8
Imperativo	8
Cliente Rest HTTP	8
Servicio Rest HTTP y Persistencia	8
GRPC	9
AMQP	10
WebSockets	10
Reactivo	11
Cliente Rest HTTP	11
Servicio Rest HTTP y persistencia	12
GRPC	12
AMQP	13
Web Sockets	13
Análisis y conclusión de Rendimiento HTTP	14
Imperativo	14
Reactivo	16
Análisis y conclusión de Gestión de recursos JVM	18
Imperativo	18
Reactivo	19
4. Conclusiones y trabajos futuros	20
Conclusiones personales del proyecto	20
Trabajos futuros	21
Pruebas de carga en otros SO y pruebas con otras herramientas	21
Pruebas de carga y rendimiento sobre el resto de tecnologías	21
Quarkus en Kubernetes con imágenes JVM y nativas	21
Bibliografía	22

1. Introducción y objetivos

¿Por qué está comparativa?

Todos sabemos que Spring Framework ha evolucionado muchísimo desde sus inicios hasta ahora, proporcionando diferentes módulos que nos ayudan y simplifican muchos de los desarrollos que se llevan a cabo actualmente. Además, con la aparición de Spring Boot, se ha hecho aún más fácil la creación de proyectos listos para correr en un entorno productivo con apenas configuración por parte del desarrollador.

Respondiendo a la pregunta de este apartado, el motivo surge como consecuencia que desde hace pocos años hasta ahora han aparecido nuevos proyectos que pretenden rivalizar cara a cara con Spring, lo cual nos parece maravilloso. Creemos que actualmente, no hay demanda de otros frameworks y esto puede ser desfavorable, ya que para cualquier desarrollo nos lanzamos a usar Spring porque nos resuelve la vida a los desarrolladores, pero no nos paramos a pensar en si la mejor solución en cuanto a la arquitectura, rendimiento o consumo de recursos.

Antes de continuar, queremos comentar que a nosotros nos encanta Spring Framework, que lo hemos empleado y lo empleamos en nuestro día a día como piedra angular en sus proyectos.

¿Por qué Quarkus?

Buena pregunta y difícil de responder, ¿por qué no Micronaut o Helidon?. Uno de los motivos es que nos gustó que el proyecto estuviera apoyado por RedHat, porque da una gran confianza en que el proyecto continúe en el tiempo, se siga desarrollando y mejorando.

Otro punto que decantó la balanza fue que Micronaut está apoyado por la gente Grails, un framework que no terminó por llegar a buen puerto y no nos terminó de convencer. También vimos por encima en la documentación, la cual está muy bien y parece muy completa, que su enfoque recuerda mucho a ciertos aspectos de Spring, lo cual no es malo, pero pensamos que si se quiere ser disruptivo y ser una alternativa a Spring hay que cambiar el planteamiento que la competencia.

Y sobre Helidon, no nos pareció que estuviera lo suficiente maduro para ser una buena elección que pueda hacer frente a Spring. Además, que esté soportado por Oracle no nos gusta, ya que puede hacerlo privativo más adelante en el futuro.

Por todos estos motivos creemos que es una buena opción Quarkus, y solo el paso del tiempo nos dirá si fue la elección acertada.

2. Introducción a Quarkus

¿Cómo ha sido el paso a Quarkus?

Redhat ha hecho muy buen trabajo en hacer que la experiencia con Quarkus sea cómoda y agradable, sobre todo para aquellos programadores que vienen de un framework tan estable como lo es Spring. Esto hace que sea realmente sencillo identificar y aplicar patrones con los que ya estamos familiarizados y que, incluso por debajo, delegan en las mismas tecnologías.

Quarkus se integra con una extensa variedad de librerías y frameworks con los que gran cantidad de profesionales ya están familiarizados. Algunas de las tecnologías integradas en Quarkus son: Eclipse MicroProfile, Context and Dependency Injection (CDI), JAX-RS, JPA, JTA, Hibernate, entre muchas otras.

Como se menciona arriba, Redhat quiere que la experiencia para los más experimentados en Spring sea lo menos dolorosa posible, y para ello Quarkus provee una serie de extensiones para varias APIs de Spring, las cuales permiten que tanto el aprendizaje como la migración de aplicaciones ya existentes y hechas en Spring tenga el menor impacto posible.

Incluso en determinados casos, una aplicación hecha en Spring podría correr en Quarkus sin cambios en código¹.

Vemos cómo, a pesar de que ambos frameworks se dirigen al mismo tipo de aplicaciones, Quarkus parte con la increíble ventaja de haber nacido aquí y ahora, prácticamente con una pizarra en blanco y que aun así es capaz de competir con los grandes.

Comunidad y documentación sobre Quarkus

La documentación disponible que existe para Quarkus es cuanto menos extensa y detallada, ya sea por parte de Quarkus, Redhat e incluso por parte de la propia comunidad.

Para empezar, vemos como cuenta una herramienta similar a *Spring initializer* (<https://start.spring.io/>), que nos permite configurar nuestra aplicación en un estado inicial con todas las dependencias que necesitemos (<https://code.quarkus.redhat.com/>), informándonos de si se encuentra soportada, si forma parte de una dependencia con *starter-code* o si se encuentra en una fase experimental del desarrollo.

¹ <https://developers.redhat.com/blog/2021/02/09/spring-boot-on-quarkus-magic-or-madness>

Por otra parte, desde la propia web de Quarkus (<https://quarkus.io/support/>), en la pestaña de *support*, tenemos las siguientes opciones disponibles:

- Propia documentación de Quarkus:
 - Get Started
 - <https://quarkus.io/get-started/>
 - Guías:
 - <https://quarkus.io/guides/>
 - FAQ:
 - <https://quarkus.io/faq/>
 - Lista de reproducción oficial:
 - <https://www.youtube.com/playlist?list=PLsM3ZE5tGAVbMz1LJqc8L5LpnfxPPKloQ>
- Tag disponible en StackOverflow:
 - <https://stackoverflow.com/questions/tagged/quarkus>
- Github discussions:
 - <https://github.com/quarkusio/quarkus/discussions>
- También cuentan con un sandbox para OpenShift:
 - <https://developers.redhat.com/articles/2021/05/31/learn-quarkus-faster-quick-starts-developer-sandbox-red-hat-openshift#>

Incluso si ninguna de estas opciones nos es suficiente, tienen abiertas listas de mensajería e incluso niveles de soporte más alto desde la propia RedHat.

Como vemos, en lo referente a soporte, RedHat y Quarkus nos dan acceso a múltiples vías de comunicación, todo enfocado a que la experiencia de los profesionales sea disfrutable y cómoda, pero sobre todo sencilla.

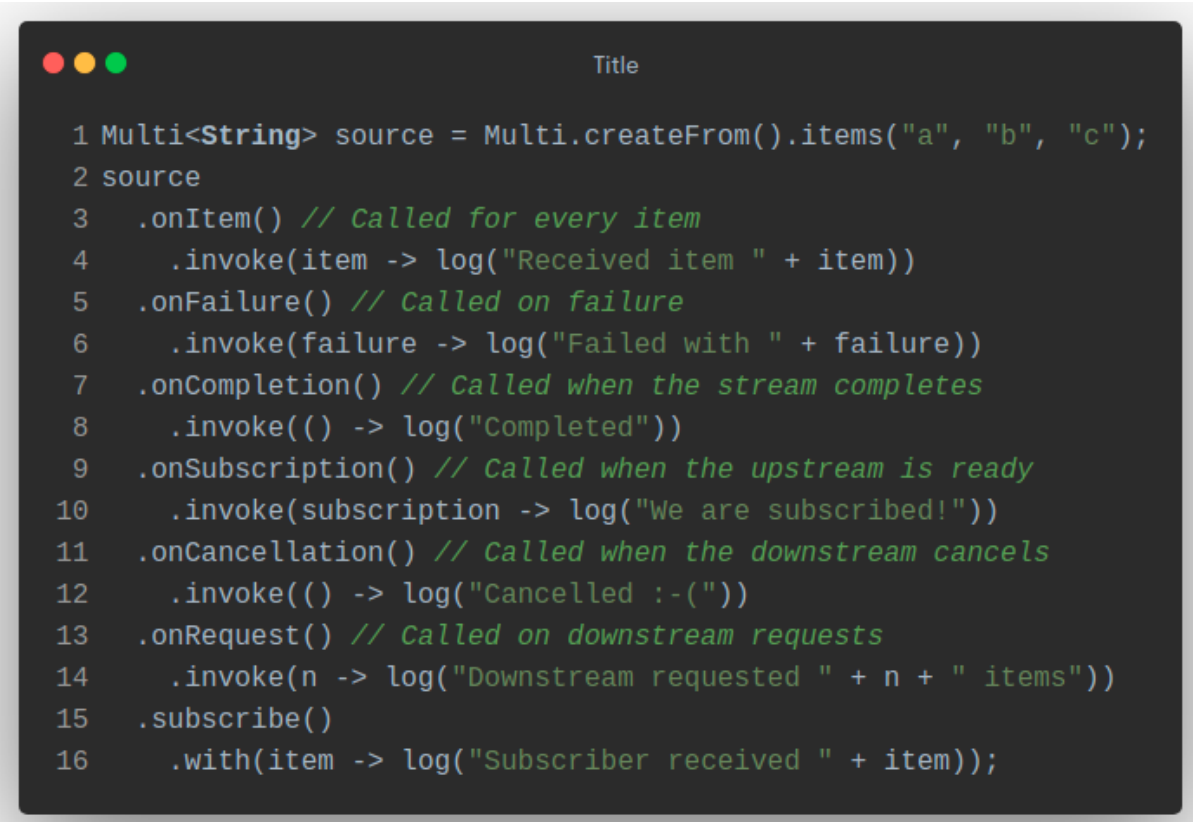
Quarkus con Mutiny

Mutiny es una librería de programación reactiva dirigida a eventos, además, es el sistema por defecto en Quarkus a la hora de crear aplicaciones reactivas.

Su modelo se sostiene sobre tres pilares fundamentales:

- Arquitectura orientada a eventos, vamos a observar los eventos que nos llegan y los vamos a manejar como queramos.
- Su API es sencilla de entender y sobre todo explícita, permitiéndonos elegir la función, método u operador que necesitemos para nuestro caso concreto.
- Solo añade dos tipos nuevos al campo de juego, *Multi* y *Uni*, con ello vamos a ser capaces de manejar cualquier tipo de interacción asíncrona en nuestro sistema.

Como se comenta anteriormente, la API de Mutiny está basada en eventos, para cada tipo de evento existe un método con prefijo “on” que nos permite manejar el evento en específico, veamos varios ejemplos:



```
1 Multi<String> source = Multi.createFrom().items("a", "b", "c");
2 source
3   .onItem() // Called for every item
4     .invoke(item -> log("Received item " + item))
5   .onFailure() // Called on failure
6     .invoke(failure -> log("Failed with " + failure))
7   .onCompletion() // Called when the stream completes
8     .invoke(() -> log("Completed"))
9   .onSubscription() // Called when the upstream is ready
10    .invoke(subscription -> log("We are subscribed!"))
11  .onCancellation() // Called when the downstream cancels
12    .invoke(() -> log("Cancelled :-("))
13  .onRequest() // Called on downstream requests
14    .invoke(n -> log("Downstream requested " + n + " items"))
15  .subscribe()
16    .with(item -> log("Subscriber received " + item));
```

Podemos observar como Mutiny nos permite decidir qué hacemos con nuestros *items*, en este caso, que va a ocurrir en caso de fallo, cuando el flujo se complete, cuando la suscripción esté lista, etc.

Respecto a los nuevos tipos que agrega esta librería, *Uni* y *Multi*, ambos son de naturaleza asíncrona, son capaces de recibir y disparar eventos en cualquier momento.

- ***Uni*** - Representa un flujo que puede contener tanto un ítem como un fallo.
- ***Multi*** - Representa un flujo potencialmente sin límites de ítems, es decir de 0 a N.

Podemos llegar a pensar que conceptualmente, un *Uni* puede ser a su vez un *Multi*, pero en la práctica, son objetos que no se usan de la misma manera, por ejemplo, si queremos manejar un valor nulo, es necesario usar *Uni*, ya que la especificación *Reactive Streams* los prohíbe en su alternativa.

Comparativa con Spring Webflux

Ambos frameworks se integran a la perfección con otros proyectos y frameworks a su vez, aunque su implementación y arquitecturas son diferentes.

Spring nos ofrece sus capacidades web en dos vertientes, una bloqueante basada en Servlets y otra no bloqueante apoyándose en WebFlux.

Por otro lado, aunque Quarkus también nos brinda esas dos variantes, nos permite usar ambos acercamientos simultáneamente, en otras palabras, el enfoque reactivo de Quarkus está embebido dentro de su propia arquitectura.

Respecto a la diferencia de tipos entre los dos frameworks, aunque cada uno se nombra de una manera como podemos observar abajo, su uso el mismo:

- **Quarkus Mutiny:** Uni y Multi
- **Spring Reactor:** Mono y Flux

En lo referente al apartado del rendimiento, en las siguientes secciones se pueden ver las diferencias más notables entre Quarkus y Spring.

Curva de aprendizaje

Como se comenta anteriormente, RedHat ha buscado que la toma de contacto con Quarkus sea sencilla y cómoda, sobre todo para desarrolladores que vengan del mundo de Spring y todos los módulos que este tiene disponibles. Su documentación es directa, explícita y sencilla de interiorizar. Es importante tener en cuenta y recordar que Quarkus ofrece muchísima compatibilidad con la API de Spring.

Para poder empezar, cualquier de los ejemplos que vienen en las guías son muy recomendables <https://quarkus.io/guides/>, ahí podremos ver cuál se acerca más al caso de uso que queremos medir, y podremos ver cómo se comporta tanto a nivel de recursos como de rendimiento.

En resumen, sus ventajas hablan por sí solas, tanto su núcleo como todos sus módulos son altamente robustos y compactos, su increíble velocidad de desarrollo, con la capacidad de activar los modos de desarrollo tanto local como dev, un ecosistema que crece a pasos agigantados, unas buenísimas guías desarrollo, documentación y comunidad, que escucha activamente y siempre agradece las contribuciones.

3. Comparativa Quarkus vs Spring

Desarrollo de aplicaciones con diferentes tecnologías

En este apartado vamos a hablar de cinco proyectos creados desde cero, utilizando tecnologías que podemos encontrar de forma más habitual hoy en día.

Estos proyectos² se han desarrollado usando la manera tanto imperativa como reactiva, con la intención de profundizar lo suficiente para exponer como de diferentes son ambos frameworks.

Imperativo

Cliente Rest HTTP

Este proyecto³ se trata de un servicio Rest sencillo que a su vez consume otro servicio rest que genera una respuesta, en este caso, un dato aleatorio sobre gatos.

Las diferencias más notorias entre ambas son que Quarkus hace uso de los componentes de JAX-RS mientras que Spring se provee del módulo de Spring MVC.

Por otra parte, respecto al cliente HTTP utilizado para hacer la llamada Rest, Quarkus trabaja con *Microprofile*, que tiene soporte nativo en este framework, mientras que en Spring nos hemos decantado por *RestTemplate*.

Como conclusión, ambas soluciones son sencillas de implementar, pero quizá el enfoque dado por Spring sea un poco más sencillo, ya que podemos crear nuestro bean y utilizarlo donde nos haga falta parametrizando de manera sencilla la url mediante propiedades.

Servicio Rest HTTP y Persistencia

Este proyecto⁴, consiste en el típico servicio Rest HTTP, el cual hace uso de una capa de persistencia para almacenar, modificar, obtener y borrar datos.

Por la parte del servicio Rest, Spring hace uso de anotaciones propias del módulo *Spring MVC*, mientras que Quarkus usa *RESTEasy*, una implementación del estándar *JAX-RS*. Con respecto al desarrollo y comprensión, no supone nada usar uno por encima del otro.

² [Comparativa de los Frameworks Spring y Quarkus sobre diferentes tecnologías](#)

³ [Comparativa Cliente Rest HTTP](#)

⁴ [Comparativa Servicio Rest HTTP y Persistencia \(Imperativo\)](#)

Sin embargo, si lo enfocamos desde el punto de vista del JAR resultante, el de Spring será más pesado, ya que usa todo un módulo entero con otras dependencias que quizá no se usen para proporcionar un servicio Rest. Quarkus por su lado tiene una librería enfocada específicamente para crear este tipo de servicio, reduciendo mucho de su JAR.

Desde el lado de persistencia, Spring se apoya en *Spring Data JPA*, un módulo que facilita la configuración de base de datos y simplifica las queries mediante la semántica de los métodos.

Igualmente, Quarkus nos proporciona herramientas para simplificar la parte de persistencia, dando la posibilidad de utilizar *Panache*, utilizando el enfoque *Active Record Pattern* o *Repository Pattern*, y además, nos da la posibilidad de utilizar Spring Data JPA para Quarkus.

GRPC

En lo que involucra a este proyecto⁵, se ha creado un servidor gRPC que, recibiendo un número en la *request*, lo multiplica por un número generado aleatoriamente entre 0 y 30 y lo devuelve en la respuesta.

Al ser gRPC, ambas implementaciones tienen una parte del desarrollo en común, es decir, nuestro fichero “generator.proto”, el fichero que contiene la definición de tu servicio, *request* y *response*.

Respecto a sus diferencias, vemos como Quarkus nos trae una de las anotaciones claves necesarias desde “io.quarkus.grpc.GrpcService”, mientras que Spring se apoya en el Spring Boot gRPC Starter⁶.

También es interesante mencionar que Quarkus trabaja de manera reactiva por defecto, cuando se crea los servicios de gRPC, como nuestro caso debía ser explícitamente no reactivo, ha sido necesario hacer bloqueante nuestro método de servicio mediante la anotación *@Blocking*, proporcionada por “io.smallrye.common.annotation.Blocking”.

En lo que respecta a Spring, hemos observado que no hay diferencias notables con Quarkus más allá de la clase necesaria de arranque y que no tiene comportamiento reactivo por defecto.

⁵ [Comparativa GRPC](#)

⁶ [Spring Boot gRPC Starter](#)

AMQP

Otra de las tecnologías empleadas, ha sido crear una aplicación⁷ basada en eventos, para ello hemos usado RabbitMQ como *broker* de mensajería. Esta aplicación genera unos pocos mensajes, que luego son consumidos y son pintados por consola

Si empezamos por el lado de Spring, este nos proporciona Spring AMQP, haciendo que la configuración y el desarrollo sea realmente sencillo. Solo es necesario crear una clase de configuración donde declaramos las *queues*, *exchanges* y *bindings* a través de Beans.

Una vez configurado estos valores, mediante “RabbitTemplate” y un método anotado con `@RabbitListener` indicando la cola, podemos emitir y consumir eventos.

En el caso de Quarkus no tiene una librería oficial, sino una creada por su comunidad, esto se debe a que la oficial está basada en una implementación reactiva.

Aquí también es necesario configurar las *queues*, *exchanges* y *bindings*, pero a diferencia de Spring, se requiere obtener la conexión a RabbitMQ y crear un canal, declarar las *queues*, *exchanges* y *bindings* dentro del canal de forma explícita. De igual manera sucede, con el Consumidor, el cual debemos instanciarlo nosotros pasando el canal creado anteriormente.

Aquí sin duda la mejor opción es utilizar Spring por su escasa configuración y su rápido desarrollo.

WebSockets

Por último, abordamos una aplicación⁸, también típica, basada en un chat entre diferentes usuarios.

Desde el lado de Spring, nos basamos en la documentación, la cual nos recomienda implementar la aplicación sobre *STOMP* y *SockJS* haciendo el desarrollo algo más elaborado.

En el lado servidor, debemos configurar un *broker* de mensajería y un tópico, y además, indicar un endpoint por donde se deben enviar los mensajes desde el lado cliente. También, tenemos que definir como unos endpoints donde mapeamos desde donde va a llegar el mensaje y a donde queremos enviar la respuesta.

⁷ [Comparativa AMQP \(Imperativo\)](#)

⁸ [Comparativa WebSockets \(Imperativo\)](#)

Por la parte cliente, tampoco es simple, se requiere instanciar un objeto *SockJS* indicando el endpoint previamente configurado en lado del servidor. A continuación, el objeto *SockJS* se utiliza para crear un objeto *STOMP*, el cual realizará las operaciones del "WebSocket", incluyendo la conexión con el *broker* del servidor.

Todas estas configuraciones y librerías adicionales, contrasta con la sencillez por parte de Quarkus. Tan solo tenemos que declarar cuál es el endpoint por parte del servidor, como si fuera un recurso REST.

Y desde el lado del cliente, únicamente se indica cuál es dicho endpoint donde debe conectarse.

Nuestra conclusión sobre esta tecnología, es que se palpa rápidamente la solución tan fresca y sencilla de Quarkus sobre la que nos proporciona Spring que da la sensación de no haber evolucionado en unos años.

Reactivo

Cliente Rest HTTP

Al igual que en otros ejemplos, este proyecto⁹ es el mismo que en el caso imperativo pero con los cambios necesarios para un enfoque reactivo.

Como principales diferencias vemos que, pesar de que Quarkus sigue haciendo uso de los componentes de *JAX-RS*, su respuesta cambia para convertirse en un objeto "io.smallrye.mutiny.Uni".

Por otro lado, Spring se comporta igual que en el ejemplo imperativo en cuanto al uso de anotaciones REST, la única diferencia es que su respuesta es un objeto de tipo "reactor.core.publisher.Mono".

Si hablamos de los cambios referencias a los clientes HTTP, vemos como Quarkus sigue haciendo uso de *Microprofile*, que tiene soporte reactivo por defecto, mientras que en Spring, en lugar de seguir usando *RestTemplate*, se ha implementado una versión reactiva de *WebClient*, que se debe de crear como una clase anotada con *@Component*, donde vamos a poder almacenar la lógica de nuestra llamada externa, el cliente reactivo de *WebClient*, como hemos comentado anteriormente, devuelve un objeto de tipo "reactor.core.publisher.Mono".

Como conclusiones, cabe destacar que ambas implementaciones son bastante similares, ya que hemos prescindido del empleo de *RestTemplate* para la parte de Spring.

⁹ [Comparativa Cliente Rest HTTP \(Reactivo\)](#)

Servicio Rest HTTP y persistencia

Para esta parte reactiva, decidimos migrar el proyecto¹⁰ imperativo existente, la aplicación hace lo mismo pero desde la filosofía reactiva.

La mayor diferencia está en las librerías, Spring usa de Spring Webflux, su API está diseñada para hacer uso del código declarativo a través de operadores y transformadores, internamente utiliza la librería Reactor basada en el *Project Reactor*. Quarkus, por otro lado, se basa en *Mutiny*, su API está orientada a los eventos y diseñada para manejarlos, lo que facilita saber qué método usar después de un evento.

Si empezamos por los endpoints Rest y nos fijamos en cómo están declarados, nos damos cuenta de que son iguales que en el proyecto imperativo, tan solo cambia el tipo de objeto que retorna y que está vinculado a la librería reactiva que emplea cada framework.

Por la parte de persistencia, también nos encontramos con alguna diferencia. La primera es que Spring ahora se apoya en *Spring Data R2DBC* para realizar transacciones reactivas, y de igual forma varía el tipo del objeto retornado un elemento reactivo.

Lo mismo ocurre con Quarkus, sigue basándose en *Panache* pero ahora con la implementación reactiva. Sin embargo, ya no dispone de la posibilidad de emplear el módulo de Spring Data R2DBC al no estar soportado aún.

GRPC

Como en casos anteriores, este proyecto¹¹ es una migración con enfoque reactivo de la versión imperativa, una app que multiplica el número recibido en la petición por otro número generado aleatoriamente entre 0 y 30.

En lo que respecta al fichero de definición de los servicios, “generator.proto”, sigue manteniéndose inmutable para los dos casos, ya que este fichero no se ve alterado sea cual sea el enfoque de la aplicación.

Como diferencias principales, la más importante es el uso del objeto utilizado para poder devolver la respuesta, en el caso de Quarkus, Mutiny nos permite aprovecharnos del objeto “io.smallrye.mutiny.Uni”, mientras que en el caso de Spring, será Reactor el que nos ceda “reactor.core.publisher.Mono”, cabe destacar que ambos objetos tienen un comportamiento muy parecido.

¹⁰ [Comparativa Servicio Rest HTTP y Persistencia \(Reactivo\)](#)

¹¹ [Comparativa GRPC \(Reactivo\)](#)

No se observan más diferencias notables en ambas implementaciones, ya que solo es necesario cambiar la respuesta. En este caso cualquiera de los dos acercamientos es una opción más que válida.

AMQP

En esta sección hemos optado por hacer otro proyecto para sacarle mayor partido al planteamiento de Quarkus. La aplicación¹² consiste en un frontal donde añades una ciudad, devuelve qué tiempo hace en mayúsculas o minúsculas.

Aquí, la configuración de Spring cambia, ahora ya no es tan simple, nos vemos obligados a crear y configurar una conexión reactiva, publicadores y consumidores, que serán los encargados de interactuar con el *broker* de mensajería.

Todo lo contrario ocurre con Quarkus, ahora sí podemos usar la librería oficial, cuyo enfoque se basa en flujos de mensajes de entrada(publicadores) y salida(consumidores), asociando los *exchanges* y las *queues* de manera muy sencilla e intuitiva.

Web Sockets

Para este ejemplo¹³, también hemos realizado otra implementación, la cual se basa en ver la cantidad de usuarios conectados a la aplicación. El motivo de este cambio de alcance se debe a dos motivos.

El primero es que la migración por el lado de Spring se complicaba, debido a que no soporta *STOMP* ni *SockJS*, como si lo hace en el lado imperativo, complicándose a nivel conceptual y de desarrollo. Ahora hay que configurar un controlador que redirija las URLs que indique a qué endpoint se conectará el WebSocket, además requiere que se extienda de una clase y sobrescriba el manejo de las sesiones conectadas.

El segundo motivo, viene por el lado de Quarkus y el planteamiento de la comunicación por la parte del servidor, el cual ya no tiene nada que ver con el enfoque imperativo. Ahora se basa en flujos de mensajes reactivos a través de canales de entrada y salida.

Por estos motivos se nos hacía complicado migrar la aplicación. Y desde el punto de vista personal, creemos que no tenía sentido.

¹² [Comparativa AMQP \(Reactivo\)](#)

¹³ [Comparativa WebSockets \(Reactivo\)](#)

Análisis y conclusión de Rendimiento HTTP

Imperativo

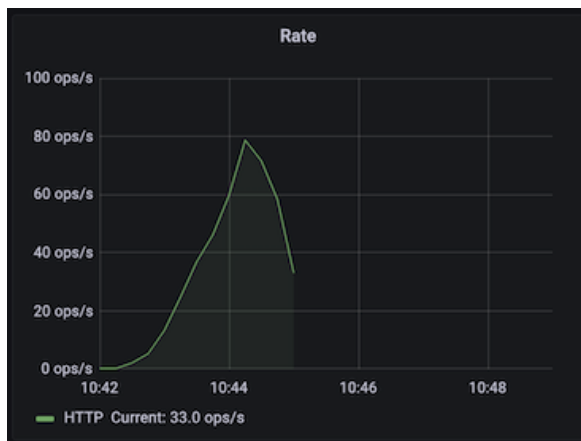
En la siguiente tabla hemos plasmado los datos de las peticiones HTTP obtenidos con Artillery en dos máquinas con diferentes recursos entre Spring y Quarkus.

	MacBook Pro 8GB				MacBook Pro 16GB			
	Spring		Quarkus		Spring		Quarkus	
	T1	T2	T1	T2	T1	T2	T1	T2
Peticiones Realizadas	60.607 (100%)	60.750 (100%)	60.736 (100%)	60.720 (100%)	60.678 (100%)	60.608 (100%)	60.668 (100%)	60.706 (100%)
Peticiones Completadas	5.844 (9,64%)	6.892 (11,34%)	60.736 (100%)	60.720 (100%)	5.802 (9,56%)	5.803 (9,57%)	52.037 (85,77%)	52.409 (86,33%)
Peticiones Fallidas	54.763 (90,36%)	53.858 (88,66%)	0 (0%)	0 (0%)	54.876 (90,44%)	54.805 (90,43%)	8.631 (14,23%)	8.297 (13,67%)
Máximas Pet/Seg	78	79	359	362	77	80	328	331

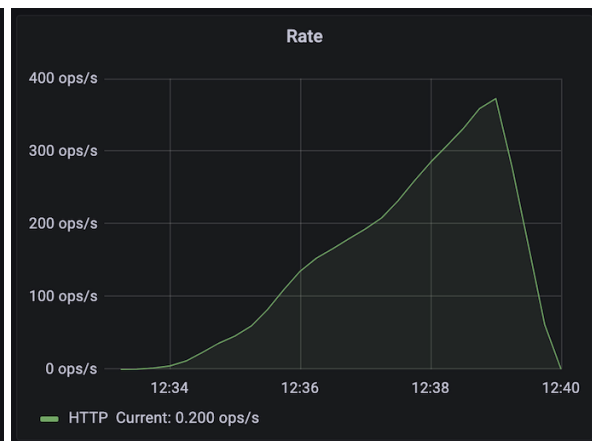
Antes de analizar los datos, se puede ver que se han realizado más o menos las mismas peticiones en todos los casos de prueba, esta diferencia se debe a [factores](#) como el tiempo de respuesta de las peticiones, latencia y rendimiento de la red. Ahora, si nos fijamos en la cantidad de peticiones completadas entre ambos frameworks vemos que hay una gran diferencia.

Por el lado de Spring, las peticiones ejecutadas con éxito son bastante más consistentes en ambas máquinas, y apenas es capaz de completar el 10% de las peticiones. Por el contrario, en una máquina Quarkus es capaz de completar 100% de las llamadas, y en la otra completa el 85% de las peticiones.

Respecto a la cantidad de peticiones máximas soportadas, podemos observar que los valores son muy iguales entre las dos máquinas. Spring no es capaz de superar las 80 peticiones concurrentes. Quarkus mientras ronda de media las 345 peticiones concurrentes por segundo, esto se traduce en que Quarkus soporta 4 veces más peticiones concurrentes.



Spring - MacBook Pro 8GB



Quarkus - MacBook Pro 8GB

Las ilustraciones de arriba podemos visualizar y comprender mejor todas las peticiones que han ido realizando e incrementando en el tiempo.

En la gráfica de Spring, observamos que nada más alcanzar el tope de peticiones, la aplicación ya no es capaz de manejar más peticiones y colapsa en su totalidad, este colapso lo vemos en el tramo donde no refleja ningún dato en sus métricas.

Todo lo contrario ocurre con Quarkus, en la gráfica se ve cómo alcanza el máximo de peticiones y luego desciende bruscamente coincidiendo cuando finaliza la prueba de carga.

A continuación, seguimos con el análisis de los tiempos de respuesta:

	MacBook Pro 8GB				MacBook Pro 16GB			
	Spring		Quarkus		Spring		Quarkus	
Tiempo Respuesta (ms)	T1	T2	T1	T2	T1	T2	T1	T2
Mediana	23,8	21,1	4	4	24,8	23,8	7	7,9
p95	308	262,5	34,1	40	327,1	308	104,4	125,2
p99	450,4	441,5	106,7	115,6	497,8	459,5	232,8	273,2

Si nos fijamos en esta tabla, nos damos cuenta que los datos obtenidos en el lado de Spring son homogéneos en ambas máquinas, sin embargo con los datos de Quarkus hay alguna mayor diferencia, pero aun así, con los peores datos de respuesta de Quarkus los datos siguen siendo bastante mejores que los mejores tiempos de Spring. Con estos datos, podemos decir con estos datos que Quarkus su mediana de tiempo de respuesta es ~4 veces más rápido, que su p95 ~5 es mejor y el p99 está ~2 veces por encima de Spring.

De todo esto, la conclusión que podemos sacar es que Quarkus tiene un rendimiento y unos tiempos de respuesta realmente buenos al lado de Spring, personalmente ninguno de los dos nos esperábamos estos resultados y mucho menos que la brecha fuera tan exagerada.

Antes de cerrar este apartado, queremos romper una lanza a favor de Spring, viendo estos resultados nos preocupamos, por lo que profundizando un poco en observamos a través de la consola de gestión y monitorización de Java, la aplicación de Spring tiene un problema con la gestión de conexiones de base de datos con Hikari. Tratamos de solucionarlo e investigamos cómo solucionar este problema pero no tuvimos tiempo suficiente para abordarlo.

Reactivo

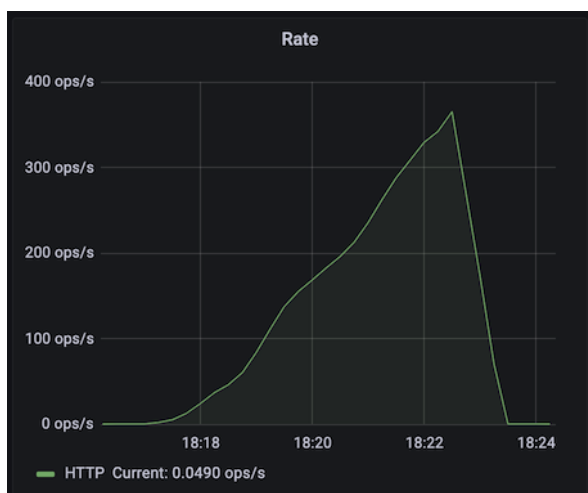
En este apartado también hemos recopilado los diferentes datos obtenidos de las peticiones HTTP con Artillery entre Spring y Quarkus, de la misma manera se han realizado las pruebas en dos máquinas con diferentes recursos:

	MacBook Pro 8GB				MacBook Pro 16GB			
	Spring		Quarkus		Spring		Quarkus	
	T1	T2	T1	T2	T1	T2	T1	T2
Peticiones Realizadas	60.634 (100%)	60.734 (100%)	60.684 (100%)	60.752 (100%)	60.619 (100%)	60.632 (100%)	60.755 (100%)	60.741 (100%)
Peticiones Completadas	60.634 (100%)	60.734 (100%)	41.602 (68,56%)	33.999 (55,96%)	52.401 (86,44%)	51.584 (85,08%)	30.704 (50,54%)	32.923 (54,20%)
Peticiones Fallidas	0 (0%)	0 (0%)	19.082 (31,44%)	26.753 (44,04%)	8.218 (13,56%)	9.048 (14,92%)	30.051 (49,46%)	27.818 (45,80%)
Máximas Pet/Seg	355	362	360	363	352	365	543	527

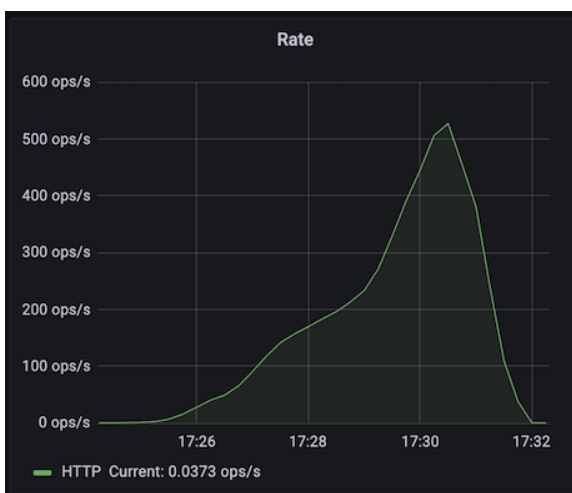
Al igual que sucede en la comparativa imperativa, el número de peticiones realizadas son muy similares en todas las pruebas.

La primera diferencia que vemos, es que se vuelcan un poco las tornas, ahora Spring completa entre un 85% y 100% de peticiones, un rendimiento mucho mejor y esperable que en la prueba imperativa, mientras que Quarkus ronda entre un 50% y un 68% de peticiones finalizadas.

Por otro lado, si miramos las peticiones máximas concurrentes en la tabla de arriba, todas ellas rondan las 350~360, excepto en la prueba del MacBook Pro 16GB, donde Quarkus llegó a soportar algo más de 500 peticiones concurrentes.



Spring - MacBook Pro 16GB



Quarkus - MacBook Pro 16GB

En estas gráficas de arriba se puede ver esta diferencia de peticiones concurrentes. A pesar de ello, la cantidad de peticiones completadas son menores que en las pruebas del MacBook Pro 8GB.

	MacBook Pro 8GB				MacBook Pro 16GB			
	Spring		Quarkus		Spring		Quarkus	
Tiempo Respuesta (ms)	T1	T2	T1	T2	T1	T2	T1	T2
Mediana	5	4	7,9	7	7	7	7	8,9
p95	80,6	48,9	90,9	96,6	96,6	92,8	100,5	115,5
p99	156	100,5	144	147	169	162,4	169	159,2

En cuanto al tiempo de respuesta son prácticamente iguales, aunque los tiempos de Spring son ligeramente mejores que los de Quarkus. Donde los tiempos para Quarkus son 1,56 veces por debajo de la mediana, el p95 es 0,67 veces inferior y el p99 es 0,18 veces peor.

Y aunque los tiempos de la mediana en Spring es casi 2 veces mejor, la diferencia sigue siendo tan inapreciable en el uso habitual de una aplicación que un usuario final no llegaría a experimentar una mala experiencia de uso.

Análisis y conclusión de Gestión de recursos JVM

Imperativo

En la tabla que vamos a ver a continuación, se pueden observar las diferencias tanto de performance como de gestión de recursos tanto de la CPU como de la propia JVM.

	Spring	Quarkus
Heap used (%)	5,76	3,5
Non-Heap used (%)	9,05	7,8
CPU Usage (%)	15,5	15,37
CPU Load	4	4
JVM Heap Committed (MB)	308	308
JVM Heap Used (MB)	236	177
JVM Non-Heap committed (MB)	129	108
JVM Non-Heap Used (MB)	118	101
JVM Total committed (MB)	417	416
JVM Total Used (MB)	350	274
JVM Memory Metaspace committed (MB)	79,7	64,5
JVM Memory Metaspace Used (MB)	77,1	63
Live Threads	43 > 213	49 > 213
Classes Loading (KB)	13,5	9,95

En primer lugar tenemos las cuatro métricas que hacen referencia a la gestión de memoria de la propia CPU, como vemos la heap usada por Quarkus es aproximadamente un 40% menor que la usada por Spring, otro punto más a favor de Quarkus en este apartado, ya que quitando la memoria *non-heap* que reduce su uso un 15% a favor de Quarkus, tanto el uso como la carga de la CPU se mantienen prácticamente iguales en ambos *frameworks*.

En segundo lugar, tenemos los recursos de la JVM divididos en diferentes grupos, los referentes a la *heap* y *non-heap*, tanto la comprometida como la que se ha usado realmente, la total de ambas y la *metaspace* memory.

Observamos cómo la diferencia de la *JVM Heap used* es de un 25%, en torno a un 16% tanto en el caso de la *JVM non-heap committed* como en la *non-heap used*, lo que se traslada en aproximadamente un 22% de diferencia en la gestión total de la memoria de la JVM a favor de Quarkus.

Respecto a la memoria *metaspace*, vuelve a ganar Quarkus con una diferencia de un 20% de media para ambos apartados.

Por último, la gestión de los hilos y las clases cargadas. En ambos casos vemos como el proceso comienza con 43 hilos en el caso de Spring y 49 en el caso de Quarkus, aunque ambos acaban esta métrica en un total de 213, la diferencia es ínfima. Las clases cargadas se sitúan en 13.5 KB en spring frente a 9.95 KB en Quarkus, una diferencia de un 17% de nuevo, a favor de Quarkus.

En conclusión, teniendo en cuenta un comportamiento bloqueante, vemos como Quarkus se anota los puntos necesarios para ganar este partido, su gestión tanto de la CPU como de la JVM es más eficiente que en el caso de Spring.

Reactivo

En este apartado, también hemos analizado su comportamiento a nivel de recursos en un caso de uso reactivo

	Spring	Quarkus
Heap used (%)	3,4	8,59
Non-Heap used (%)	8,44	6,63
CPU Usage (%)	22,99	21,7
CPU Load	4	4
JVM Heap Committed (MB)	308	444
JVM Heap Used (MB)	193	352
JVM Non-Heap committed (MB)	112	92,1
JVM Non-Heap Used (MB)	107	86,9
JVM Total committed (MB)	420	536
JVM Total Used (MB)	291	436
JVM Memory Metaspace committed (MB)	63,8	56,4
JVM Memory Metaspace Used (MB)	61,8	54,8
Live Threads	11 > 17	16
Classes Loading (KB)	11	8,83

Si analizamos el primer apartado de la tabla, vemos como Spring hace una mejor gestión de la *heap* usada, suponiendo una diferencia de casi un 60% respecto a Quarkus.

Respecto a la JVM, vuelve a ganar Spring, a pesar de que Quarkus hace una gestión más eficiente de la *non-heap*, en torno a un 20%, en el resto de apartados se queda atrás, suponiendo casi un 35% en el uso total de la memoria de la JVM, a favor de Spring.

En referencia a los apartados de la memoria *metaspace*, Quarkus se anota otro tanto con casi un 12% de diferencia en la memoria *metaspace* usada.

Para terminar, la cantidad de hilos en Spring ha crecido, comenzando con 11 y acabando con 17, mientras que en Quarkus, se ha mantenido estable en 16 hilos. La diferencia de clases cargadas en memoria es parecida al caso imperativo, suponiendo un 19% aproximadamente a favor de Quarkus.

Para este caso de uso, parece que Spring hace una mejor gestión de determinados apartados de la CPU y sobre todo de los recursos de la JVM.

4. Conclusiones y trabajos futuros

Conclusiones personales del proyecto

Como ya comentamos al principio, somos desarrolladores que conocen de primera mano lo que es trabajar con Spring, conocemos bastante bien los beneficios y facilidades que aporta trabajar bajo este ecosistema. Es verdad que Quarkus no se puede comparar aún en todos los aspectos con Spring debido a esta recién salido del horno, pero si nos ha dejado muy buenas sensaciones.

Nuestra experiencia de pasar de Spring a Quarkus ha sido muy positiva, no ha sido para nada complicado, podríamos decir que nos ha parecido más sencillo adoptar los conceptos básicos en los que se basa porque usa estándares, como la inyección de dependencia se basa en CDI, anotaciones JAX-RS para endpoints REST, etc.

En el repositorio hemos abordado una comparativa donde se hace un análisis de las diferencias entre Spring y Quarkus con diferentes tecnologías: endpoints y clientes REST, GRPC, base de datos relaciones (ORM), Websockets y broker de mensajería (AMQP).

Hemos abordado estas tecnologías y no otras porque son las que un desarrollador se puede encontrar más frecuentemente en el mundo laboral. Y durante los

desarrollos y análisis entre proyectos, hemos observado como Quarkus provee un enfoque y una implementación distinta pero a la vez sencilla, limpia e intuitiva.

Si lo miramos desde el punto de rendimiento, previamente hablado en la sección 3. Comparativa Quarkus vs Spring, Quarkus consigue buenos resultados si pensamos que no tiene la madurez que tiene Spring. Creemos que es cuestión de poco tiempo que Quarkus comience a pulirse para que vaya mejor de lo ya va.

Por todos estos motivos, aunque aún haya algunas cosas por hacer y por probar, creemos que Quarkus es una excelente opción para empezar a aprender y con el que construir proyectos, sobre todo en aquellos que están pensados para estar en contenedores ya que es ahí donde reside toda su filosofía.

Trabajos futuros

Pruebas de carga en otros SO y pruebas con otras herramientas

El motivo de extender la prueba a otros Sistemas Operativos es ver cómo se comportan las aplicaciones, y comparar el rendimiento con los datos mostrados aquí.

Además, creemos que un buen método para corroborar que los datos obtenidos por Artillery son correctos, es hacer más pruebas de carga con otras herramientas y ver que los datos están alineados con los obtenidos en estas pruebas. Nuestra propuesta es hacer las mismas pruebas de carga con otras herramientas como K6, con una configuración algo diferente pero igual o más flexible, o usar wrk2, una herramienta de carga constante sencilla de usar, menos flexible pero potente.

Pruebas de carga y rendimiento sobre el resto de tecnologías

Actualmente, no hemos podido abordar las pruebas para WebSockets, GRPC y AMQP, por lo que sería de gran ayuda poder conocer cómo se comportan cuando están sometidas a una gran carga y cómo esto afecta al rendimiento y consumo de recursos de las mismas.

Quarkus en Kubernetes con imágenes JVM y nativas

Uno de los puntos fuertes que tiene Quarkus es su rapidez para levantar las aplicaciones, por lo que desplegar las imágenes docker en un clúster de Kubernetes y someter la aplicación a una carga elevada para que escale y ver el rendimiento de esta sería otro punto para analizar con Spring.

Además, este mismo escenario habría que aplicarlo con imágenes nativas y observar cómo mejora el rendimiento en comparación con las imágenes docker propias de Quarkus.

Actualmente, Spring ofrece la creación de imágenes nativas y aunque no hay una versión estable, se podría realizar otro análisis entre las imágenes nativas de cada framework.

Bibliografía

- <https://quarkus.io/guides/>
- <https://docs.spring.io/spring-framework/docs/current/reference/html/>
- <https://smallrye.io/smallrye-mutiny/>
- <https://smallrye.io/smallrye-reactive-messaging/3.16.0/>
- https://prometheus.io/docs/prometheus/latest/getting_started/
- <https://grafana.com/docs/grafana/latest/installation/docker/>
- <https://www.artillery.io/docs>
- <https://quarkus.io/guides/grpc-getting-started>
- <https://quarkus.io/guides/rest-client>
- <https://quarkus.io/guides/getting-started-reactive>