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

Curso académico 2021/2022

Trabajo de Fin de Máster

Alternative Frameworks

Autor: Joaquín Antonio De Vicente López
Tutor: Micael Gallego Carrillo

Resumen

Los frameworks constituyen herramientas muy potentes que permiten resolver problemas recurrentes en el mundo del software. Bajo esta premisa se encuentra este proyecto en el que se tratan frameworks alternativos a los vistos en el máster: Quarkus y NestJS

En esta memoria se describe el contexto tecnológico, el estado del arte, la motivación, las principales características, objetivo y limitaciones que se han encontrado en el desarrollo de este proyecto.

El desarrollo de este proyecto se encuentra dividido en dos partes, por un lado esta memoria que describe lo anteriormente mencionado y por otro lado los ejemplos en los dos frameworks mencionados que se encuentran en el repositorio de Git indicado en los anexos. Estos ejemplos son en algunos casos, idénticos de los vistos en el máster, en otros similares por limitaciones técnicas, en otros ampliados y en otros simplificados bien porque el framework otorga dicha facilidad bien por limitaciones. En otros casos, puede decirse simplemente que son distintos.

Estos ejemplos son en algunos casos ejemplos básicos que no requieren de ninguna documentación adicional salvo como se ejecutan mientras que en los casos más complejos se ofrecen colecciones Postman y descripciones más detalladas sobre su ejecución. No obstante según lo analizado en esta memoria y lo visto en el máster, cualquier otro alumno debería ser capaz al momento de finalización el más entender los ejemplos propuestos y ser capaz de trasladar conceptos, ideas y tecnología de unos frameworks y lenguajes a otros.

En este contexto, se plantean como alternativos estos frameworks para que el alumno sea capaz de lo anterior y reforzar y aprender nuevos conceptos ya que este se trata de un trabajo puramente didáctico.

Lo que podrá encontrar en las próximas páginas es el detalle sobre los conceptos sobre los que subyacen estos frameworks así como algunos de los problemas encontrados al realizar estos ejemplos y en el repositorio adjunto en los anexos los ejemplos realizados.

Índice

Capítulo 1: Introducción

En este capítulo, se hablará del contexto en el que ha sido planteado este proyecto, de las principales motivaciones que han dado lugar a su desarrollo y de los objetivos marcados así como de las contribuciones realizadas. En él se introducirán los frameworks empleados en el desarrollo de este trabajo y se explicarán brevemente los antecedentes y el contexto tecnológico en el que se encuentran.

1.1 Contexto y Motivación

En los últimos años, las aplicaciones que se desarrollan han ido evolucionado de una velocidad de desarrollo lenta, poco flexible, propensa a errores y monolítica hacia una forma de desarrollo y despliegue cada vez más automatizada, rápida y sencilla; esto a su vez ha disminuido el número de problemas técnicos y el tiempo de recuperación ante incidentes. La razón por la que se produce esto es por la automatización del desarrollo de Software, que automatiza además otras cosas, el diseño de su arquitectura, la realización de pruebas, su entrega y/o despliegue.

En este contexto exclusivo del desarrollo del Software se encuentran los frameworks, marcos de trabajo que constituyen herramientas potentísimas para resolver problemas recurrentes del desarrollo de Software. Su mecanismo de funcionamiento es el siguiente: proporcionan al desarrollador una serie de elementos básicos u operaciones provenientes de distintos módulos o librerías propios del framework que bien por sí mismos, bien en consonancia con otros, aplican un patrón, el que corresponda al problema en cuestión, para resolverlo; y lo hace sin que el usuario desarrollador sea consciente de cómo lo consigue, funciona como una caja negra. Es decir, el desarrollador desconoce cómo el framework resuelve el problema a bajo nivel, simplemente le ofrece una primitiva para resolverlo y le brinda al desarrollador dicho mecanismo para aplicarlo. No obstante, aunque a priori no sepa cómo lo hace, podría llegar a saberlo siempre que tenga acceso al código del framework o este sea Open Source. Uno de ejemplos de estas primitivas es el uso de la Inyección de Dependencias, la Programación Orientada a Aspectos o el uso de anotaciones para definir e instanciar elementos transversales en una aplicación como Entidades de Bases de Datos, Controladores REST, Controladores Web, Repositorios de Base de Datos, Servicios y Protocolos, Configuraciones, etc.

Dentro de los frameworks es posible comprobar que existe una variedad inmensa de ellos y que las tecnologías y lenguajes de programación que emplean son muy variadas. Esto llevaría a pensar a uno que el conocimiento necesario para poder usar uno u otro es extremadamente alto y que la curva de aprendizaje para sentirse cómodo programando con un framework es muy pronunciada, pero nada más lejos de la realidad, los frameworks cada vez ofrecen primitivas que son más parecidas entre ellos y entre los diferentes lenguajes. Al contrario de lo que podría pensarse estas primitivas deben ser más sencillas y efectivas que las de su competencia y la curva de aprendizaje más corta si los que mantienen y evolucionan el framework quieren que los desarrolladores o empresas tengan verdaderos motivos para usarlo, ya sean en un nuevo aplicativo, migrando una aplicación de un framework a otro o de una aplicación que simplemente no usaba ninguno.

Dentro de este contexto, como se puede observar la razón para usar framework es evidente pero el motivo para decantarse por uno u otro es un tema complicado y no es trivial. Como bien es sabido, la empresa o los desarrolladores que mantienen y evolucionan un framework deben conocer el estado del arte de los frameworks en el mundo del Software y tener amplios conocimientos sobre ellos para proporcionar primitivas mejores que su competencia en el mismo lenguaje o incluso en distinto si verdaderamente quieren permanecer y no caer en el olvido. Un criterio razonable es utilizar un framework que use uno de los lenguajes soportados por tu empresa o grupo de trabajo, que sea lo suficiente generalista para poder llevarlo a otros desarrollos pero también lo suficiente específico para resolver problemas que otros frameworks no pueden.

Si bien es cierto que hay mayor cantidad de frameworks en aquellos lenguajes dedicados al desarrollo de aplicaciones o servicios empresariales como aplicaciones Web, servicios REST, etc, el concepto de framework no es exclusivo de ello, también hay frameworks dedicados a la investigación y a la Inteligencia Artificial entre otros. No obstante en este trabajo nos centraremos en los primeros y más específicamente en dos que guardan relación con el propio contenido del máster y a los que consideramos alternativos: Quarkus y NestJS. De los dos frameworks anteriores cada uno se corresponde con un marco de trabajo o lenguaje homólogo visto en el máster. En el caso de Quarkus, que está escrito en Java, su homólogo es Spring. En el caso de NestJS, que está escrito en TypeScript y se ejecuta sobre el mismo entorno de ejecución, Node.js, sus homólogos son JavaScript y Express. Ambos frameworks tiene particularidades diferentes que se detallarán más adelante pero la motivación principal por lo que se han elegido es por estar enfocados principalmente en resolver la problemática con la que versa el propio título del máster, el Desarrollo y Despliegue de aplicaciones Nativas en la Nube.

Esta problemática, a la que no es ajena ninguna empresa hoy en día, consiste en disponer de mecanismos de desarrollo y despliegue, aunque también de prueba, de una aplicación en la nube para que podamos decir que es nativa de la nube. La razón por la que tiene que ser nativa de la nube es porque el contexto tecnológico en el que vivimos, la reducción de los costes del Hardware, el pago por uso, la descentralización, la automatización del uso de la infraestructura y el despliegue de Software hace que desarrollar un Producto Software debe ser lo suficientemente versátil como para que sea agnóstico de la Infraestructura si quiere permanecer o adaptarse en un mercado cada vez más cambiante

Llamamos nube pues, a esta Infraestructura Hardware sobre la que el Software debe ser agnóstico de forma que pueda ser desplegada en otra sin cambios ni mayor dificultad ya que sino el Software conocerá los detalles de esa Infraestructura y no estará preparado como para desplegarse en otra o no será lo suficiente resiliente para permanecer desprovista de problemas mediante una u otra tecnología en caso de que esta quede obsoleta o sea desplazada por una mejor. Normalmente una aplicación nativa de la nube es más resistente a desplegarse en cualquier sitio que una que no. Las aplicaciones nativas de la nube es más probable que tengan la capacidad de ser desplegadas On Premise que las aplicaciones On Premise sean desplegadas en la nube, pero una cosa es la capacidad y otra es la necesidad. La propensión si se quiere que el Producto Software prospere es a que sea del revés, pero para ello los equipos de desarrollo y el Software deben estar preparados. Una manera es mediante el uso de frameworks como los que se tratan en este trabajo.

1.2 Objetivos

El principal objetivo de este proyecto es entender cómo funcionan otros frameworks para resolver los mismos problemas o problemas similares y como sus primitivas, aunque distintas suponen alternativas igual de válidas para resolver los mismo problemas de desarrollo. Para ello se proporcionan los ejemplos del módulo II del máster, Servicios web: tecnologías, arquitecturas, pruebas y persistencia, resueltos en estos dos frameworks.

El porqué solo de este módulo es porque es el que trata de mejor manera los problemas recurrentes del software desde un punto de vista de los frameworks, mientras que en el módulo I se trata desde un punto de vista teórico con los patrones de diseño, los tipos de arquitecturas, las metodologías de desarrollo y en el módulo III y IV desde un puto de vista de Despliegue y Operaciones, haciendo hincapie en la nube y en DevOps pero no tanto en la parte de desarrollo.

Cabe destacar que algunos ejemplos han sido modificados para añadir complejidad bien sea para hacer más interesante el ejemplo y en otros para quitarla y mostrar cómo este framework alternativo simplifica aún más el problema, en otros casos ni siquiera se ha buscado esto y es porque simplemente la naturaleza de resolverlo es más sencilla y en otros simplemente había limitaciones del framework por lo que se tenía que abordar el ejemplo de otra manera. También, como es evidente hay ejemplos que no tiene sentido ser traídos a los framework al tratarse de librerías o programas comunes cuyo ejemplo es agnóstico del framework.

El principal objetivo pues, es didáctico, se trata de que el alumno vea el código, entienda los ejemplos y sepa percibir, sin conocer los detalles del framework las diferencias y similitudes con sus homólogos. A partir de ahí el alumno puede profundizar y estudiar estos frameworks alternativos más en detalle y compararlos con los que ya conoce. Estudiar sus diferencias, indagar y buscar ejemplos similares con otros que no conoce e incentivar a que descubra nuevos y saque sus propias conclusiones. De este objetivo principal se derivan dos objetivos secundarios:

- Que el alumno alcance las aptitudes y el dominio necesarios para que sea capaz de implementar ejemplos similares y nuevos en un framework que no conoce pero en un lenguaje que conoce o domina.

- Que el alumno alcance las aptitudes y el dominio necesarios para que sea capaz de implementar ejemplos similares y nuevos en un framework y en un lenguaje que ni conoce ni domina.

El objetivo último es por lo tanto, que el alumno adquiera las aptitudes necesarias para ser capaz de desarrollar ejemplos de aplicaciones y Software enfocado a los Servicios Web en cualquier lenguaje y framework siempre que se le disponga de las herramientas necesarias para ello.

1.3 Contribuciones

La principal contribución que se realiza con este proyecto es la implementación de los distintos ejemplos del módulo II en los frameworks alternativos mencionados anteriormente. Esta contribución garantiza la consecución de los objetivos y sirve de referencia para futuros trabajos en otros frameworks en caso de que tal cosa se plantee. También puede servir de material auxiliar de referencia del propio máster en caso de que el director de este así lo decida.

Esta contribución es ampliada por esta memoria en el desarrollo de su contenido en los Capítulo 2 y 3. En ellos se tratan ambos framework desde el punto de vista de su origen, su contexto tecnológico, las limitaciones que tienen, los errores más frecuentes, sus particularidades, su finalidad, sus detalles técnicos, competidores y los objetivos que tiene marcados.

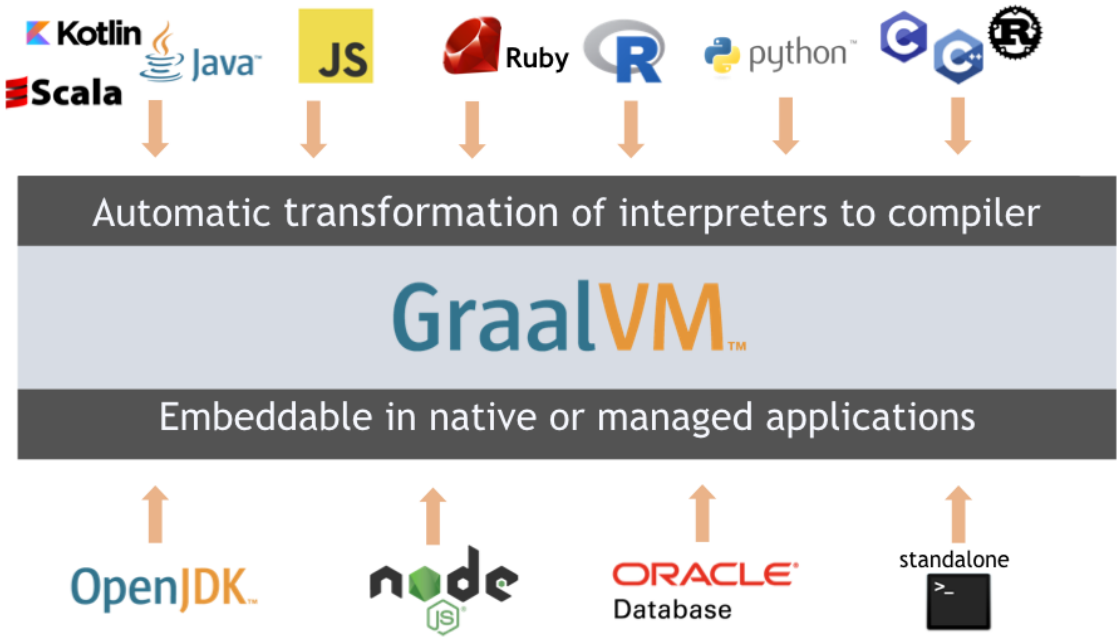
Dicha contribución ampliada no constituye un material didáctico del nivel y rigor del contenido del temario del máster ya que en él se tratan las cuestiones técnicas acerca del software y los frameworks con mucho más detalle y de forma mucho más extensa. Esta memoria constituye a grandes rasgos una comparativa de estos frameworks alternativos con sus homólogos de forma que cualquier alumno pueda lograr alcanzar los objetivos establecidos, pero no tiene por intención u objetivo asemejarse a la calidad de dicho material didáctico porque se encuentra fuera de su alcance. Esta memoria, además de esta comparativa, describe, al margen de su complejidad, un estado del arte de ambos frameworks.

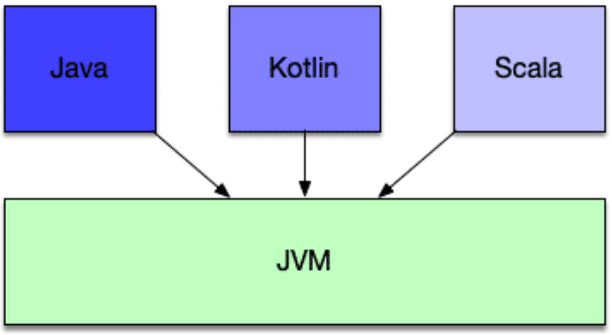
Capítulo 2: Quarkus

2.1 Estado del Arte

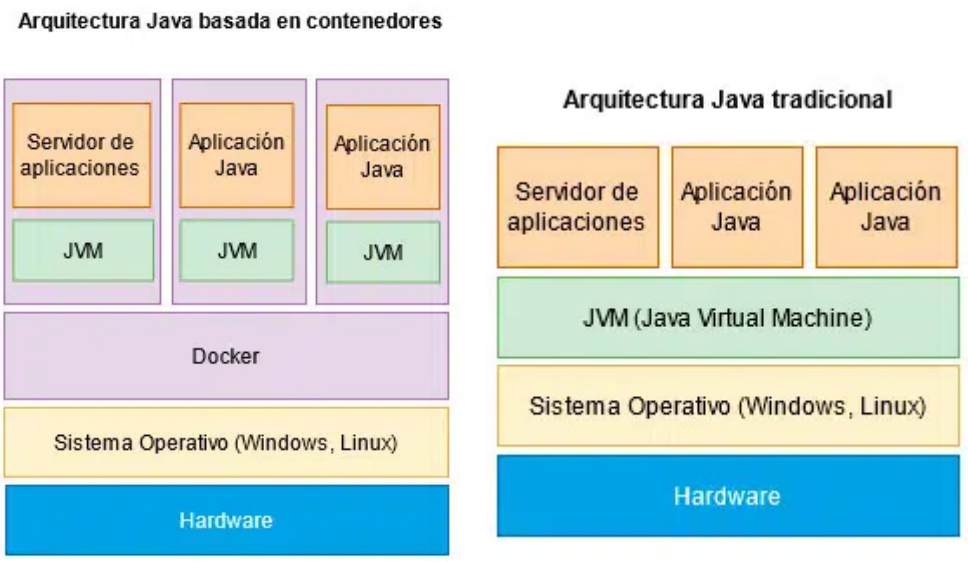
Quarkus es un framework de desarrollo de Aplicaciones Java creado por RedHat el 20 de Marzo de 2019. A fecha de hoy, cuenta con menos de 4 años de desarrollo a sus espaldas por lo que podemos decir que es un framework bastante reciente. Quarkus está enfocado en el desarrollo y despliegue de Aplicaciones Java en contenedores que arrancan rápido y consumen poca memoria y sean compatibles con el paradigma de computación distribuida en microservicios y Serverless por lo que está ampliamente integrado con Kubernetes y Funciones como Servicio o FaaS.

La manera en la que logra esto es mediante la compilación anticipada AOT (Ahead-Of-time compilation) de los fuentes a través de la Máquina Virtual de Java (JVM) GraalVM en una imagen nativa de manera que para su ejecución en un pueda prescindir de la JVM. Esto hace que la imagen sea menos pesada y arranque más rápidamente por lo que lo hace idóneo para los paradigmas mencionados anteriormente. Es por eso que se dice que Quarkus es un framework nativo de la nube. Respecto a GraalVM, esta es una JVM políglota basada en la JVM Hotspot y en OpenJDK. A diferencia de otras JVM que se integra sólo con lenguajes como Kotlin y Scala esta se integra con muchos otros lenguajes. A continuación se muestran dos imágenes con los lenguajes en los que se integran ambas JVM.





A continuación también puede observarse la diferencia entre una JVM tradicional y una JVM que corre sobre Docker. En cualquier caso no es mejor que una imagen nativa corriendo sobre Docker como veremos más adelante.



2.2 Caracaterísticas

CDI

Quarkus ofrece la Inyección de Dependencias y Contextos (CDI) a través de distintas funciones que se definen en la Jakarta CDI o JCDI de Java Enterprise Edition, también conocida como JEE. Si bien es cierto que su especificación no es una implementación completa de esta es lo suficiente completa para desarrollar los mismos ejemplos. De hecho en algunos casos al ser más simplificada facilita la cantidad de anotaciones que el desarrollador tiene que aprender y la manera en la que se inyectan los Beans. Algunos ejemplos son:

- El archivo Beans.xml es ignorado.
- Las Clases que crean Beans se anotan con @Dependant
- La Creación de Beans se produce con @Produces
- La inyección de Beans se realiza con @Inject
- La clase que va a ser inyectada debe estar anotada con @Singleton o @ApplicationScoped. Si bien es cierto que Singleton realiza una inyección entusiasta y ApplicationScoped realiza una inyección perezosa, Singleton no puede usar para realizar mocks con QuarkusMock. La razón por la que es más rápido con Singleton es porque es directa mientras que con ApplicationScoped hay una Clase proxy que es el realiza su inyección.

Si analizamos las diferencias de las anotaciones con respecto a Spring vemos que queda bastante simplificado. En el caso de propiedades definidas en el application.properties empleamos @ConfigProperty. @Scope dependería del caso aunque yo recomiendo emplear @ApplicationScoped incluso en los ejemplos que la siguiente imagen muestra como @Singleton porque a la hora de realizar mockeos es menos problemático ya sea con QuarkusMock o con otra librería como mockito. Un ejemplo muy claro de esto es que si los clientes REST, que se definen de manera declarativa como hace Feign si no se anotan con @ApplicationScoped no podrán ser mockeados. Por último @ComponentScan que no aplicaría.

Spring	CDI / MicroProfile
@Autowired	@Inject
@Qualifier	@Named
@Value	@ConfigProperty
@Component	@Singleton
@Service	@Singleton
@Repository	@Singleton
@Configuration	@ApplicationScoped
@Bean	@Produces
@Scope	No hay correspondencia directa. Dependiendo del valor de @Scope, se usará @Singleton, @ApplicationScoped, @SessionScoped, @RequestScoped o @Dependent según se necesite
@ComponentScan	No hay correspondencia. Quarkus examina todo el classpath en tiempo de construcción.

Continous Testing

Quarkus permite la posibilidad de realizar test continuo y en caliente una vez arrancada la aplicación, para ello define la propiedad quarkus.test.continuous-testing que puede tomar valores, paused, enabled o disabled, lo que significa que al arrancar la aplicación no se ejecutarán o sí, o que solo cuando el usuario los resuma volverán a lanzarse. Además cuando Quarkus se ejecuta en modo develop funciona con live reload lo cual es muy útil cuando se desarrolla para detectar fallos rápidamente. Si bien es cierto, estas dos características no funcionan cuando se ejecuta la aplicación en modo terminal con @QuarkusMain como si fuera una aplicación Java de toda la vida. En el caso de los tests porque no los encuentra y el live reload porque simplemente no funciona.

Controladores

En Quarkus los Controladores se llaman Resources y no hacen distinción entre Controladores REST o Web. Las anotaciones principales para dichos métodos son

@Path para definir el endpoint por el cual entrará el método al llegar la petición. Puede definirse a nivel de clase y dentro de cada método de forma que se va concatenando hacia dentro.

@GET @POST @PUT @DELETE @PATCH en mayusculas para los distintos verbos HTTP.

Response del paquete javax.ws.rs.core.Response para las respuestas. Aunque puede devolverse el objeto sin más y ya se encargará jackson de convertirlo en respuesta HTTP con valor 200 por defecto, Response permite definir el HTTP Status y otros headers como el location.

@Inject
HttpSession del paquete javax.servlet.http: Los objetos HttpSession se declaran al principio de la clase y no pertenecen a la petición.

@Context
HttpRequest del paquete org.jboss.resteasy.spi para poder obtener la url de la propia petición

@PathParam(“id”) si el Path tiene un id de nombre “id”. Por ejemplo @Path(“/{id}”)

@BeanParam sobre el objeto dentro de la cabecera del método sobre el que se realiza la petición y @FormParam sobre los campos del objeto que formarán parte de dicho formulario del formulario. Esto es cuando acepta peticiones de formulario con los campos codificados en la URL: x-form-urlencoded. Si no se define nada el Bean las peticiones y respuestas lo trataran como un JSON y lo serializarán o deserializarán con la definición de dicho Bean.

Qute Templating Engine

Quarkus utiliza un motor de templating propio por lo que no depende de ninguna librería externa como mustache o Handlebars. La razón de esto es porque necesitaba algo que encajasen con sus necesidades y que soportará tanto un modo imperativo como reactivo e hiciera poco uso de la reflexión o Java Reflection. Su sintaxis no es muy distinta de mustache.

A diferencia del templating en Spring que puede utilizar librerías propia como mustache, los controladores Web se definen igual que un controlador Rest per devuelven una instancia de la plantilla con los parámetros pasados de igual forma que lo hacía un ModelAndView en Spring con la función data en lugar de setAttribute. Además hace uso de Interfaces fluidez por lo que permite setear varios objetos en una solo línea antes de instanciar la plantilla por lo que ocupa muchas menos líneas código y es más práctica. Cabe destacar que resulta aconsejable retornar con @Produces o @Consumes el MediaType en los Controladores MVC.

Reactividad

Quarkus permite la posibilidad de trabajar de manera reactiva a través de la librería SmallRye Mutiny y Vert.x. Esto le permite reaccionar a eventos de forma asíncrona

en lugar de bloquearse y terminar de responder lo cual penalizaría mucho el rendimiento. Esto hace que Quarkus sea idóneo para una arquitectura de MicroServicios orientada a eventos. Distintas librerías y extensiones como las de BBDD, mensajería, colas tiene su variante asíncrona con la salvedad de que el cliente de Base de Datos, H2, es el único que no tiene variante reactiva,

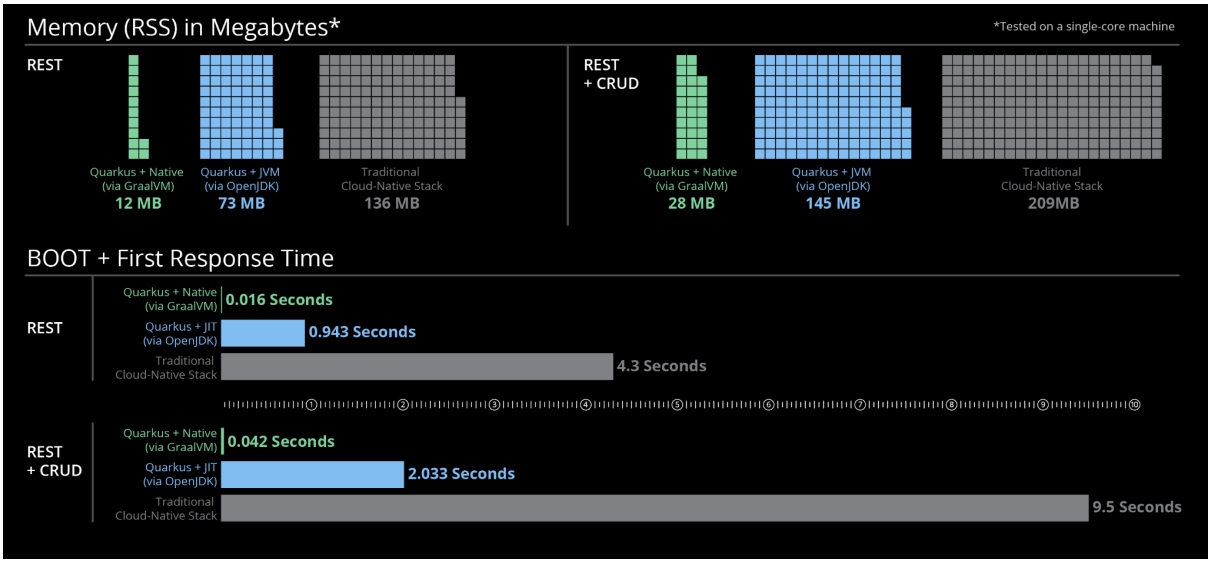
Construcción de Imágenes

Quarkus permite la creación de imágenes de distinto tipo con herramientas de compilación y empaquetado como Jib que pertenece a Google, Buildpacks que pertenece a Paketo aunque existen otras alternativas como Heroku Buildpacks pero Quarkus no se integra con ella, Docker y S2I que es un framework de construcción de imágenes. En todas ellas se crea una imagen con el tipo de empaquetado definido en la propiedad quarkus.package.type. Que si no se incluyen dicha propiedad tomará el valor de jar por lo que construirá un fast-jar. Es decir creará un jar no nativo que será más pesado pero se construirá más rápido. Luego hay otras opciones como legacy-jar para releases anterior a la 1.12, uber-jar y native.
De estas dos últimas la imagen nativa se trata de una imagen que se adapta al sistemas host y que es más lenta de construir pero menos pesada y consume menos recursos y memoria a la hora de ejecutarse ya que no requiere de Máquina Virtual de Java. En el caso del uber-jar se trata de un jar pesado o fat jar, similar a los paquetes .war que contiene todas las clases y dependencias pero que consume más recursos y es más lento. Tiene la ventaja de que al tener todas las dependencias puede ser ejecutado con el Class Loader por defecto de Java. Por el contrario, mismas rutas de ficheros en distintos JAR se sobrescribirán unos a otros, esto es lo que se conoce como unshaded y para solucionar esto se puede que apuntes esto hay que utilizar el shade plugin de maven para sombrear o enmascarar las rutas y que no se pisen entre ellos. Otra opción es generar un Jar de Jars para que no colisiones las rutas ni clases o dependencias. El problema es que para solucionar esto hay que seleccionar un Class Loader especifico para arrancar la aplicación.

En los ejemplos de construcción del tema 5 se muestran ejemplos de imágenes construidas para docker en las distintas tecnologías: jib y docker en imágenes de tipo fast-jar y native. En el caso de buildpacks no ha sido posible porque a fecha de hoy en hay una implementación oficial de buildpacks para aplicaciones que se ejecuten en Quarkus y los ejemplos de buildpacks propios no funcionan con Java 17. Respecto al otro método de construcción s2i, no ha sido posible porque se requiere de la instalación de Kubernetes en OpenShift, la cual no es gratuita.

Algunas de estas soluciones como jib o docker parten de una imágen base distinta para generar la imagen final. Esta imagen puede configurarse para bajarse en docker o en podman siempre que se tenga instalado pero la imagen final siempre será en docker. En el caso de jib hace usos de una imágen de la GraalVM que genera la imagen final ya sea o no nativa. En el caso de docker, si es fast-jar o native utiliza la Universal Base Image de RedHat ubi9/ubi-minimal:9.1.0 y genera el paquete nativo o no. En el caso distroless, que evidentemente es fast-jar emplea o una imagen propia de Quarkus: quarkus-micro-image:2.0

A continuación se muestra una imagen de diferencia en el rendimiento entre una imagen nativa sobre GraalVM y una que no:



Integraciones

Quarkus se integra con otras librerías conocidas como: restEasy, openapi, jackson, jsonb, RestEasy, microprofile, Smallrye, Kafka, Graphql, gRPC, testContainers, etc. A continuación se describen alguna de las características de estas integraciones:

- En el caso de Graphql dispone de un PlayGround en q/graphql-ui/ para realizar pruebas de la misma forma que lo harías en Postman, además permite descargarte el schema.

- Quarkus define un Client de Graphql declarativo al igual que con el cliente REST y otro dinámico con su propia sintaxis para Graphql.

- Dependiendo de que que procesador de JSON uses junto a restEasy podrás tener o no ciertos problemas debido a la serialización y al uso de anotaciones como @JsonView, @JsonIgnore, etc.

- La implementación de la extensión de Kafka utiliza en verdad RedPanda, lo cual es un competidor que se ha puesto muy de moda y para que es mucho más rápido que Kafka.

- En Base de Datos y mensajería por colas si no defines un datasource url definido, Quakus usará el de por defecto y hará uso test containers para bajar la imagen si es que no está ya construida y arrancará el contenedor siempre que tengas docker instalado.

- Des la ruta local en q/dev/ permite editar propiedades, visualizar los Beans y su estado, Interceptores, Observers, Eventos disparados, servicios como gRPC, las Entidades de BBDD, las Named Queries, reiniciar la BBDD, etc.

2.3 Limitaciones y Problemas Encontrados

En comparación con NestJS dónde prácticamente no me he encontrado ningún problema, en Quarkus han surgido bastantes problemas, algunos resueltos, otros con workarounds como solución temporal. A continuación algunos ejemplos

Ausencia de Implementación oficial en BuildPacks:

Durante la ejecución del ejemplo de buildpack, no había código en el repositorio de Paketo y solo una Pull Request, ahora sí, pero aún no se ha integrado con el repositorio de Quarkus en ninguna release.

Estado: Cerrado, pendiente de cómo evoluciona el desarrollo sobre repositorio de Paketo y responde Quarkus con el buildpack oficial.

Repositorios:

quarkusio/quarkus-buildpacks
A Cloud Native Buildpack that enables easy builds for Quarkus applications

El Work In Progress quarkusio/quarkus-buildpacks no funciona con Java 17.

A Cloud Native Buildpack that enables easy builds for Quarkus applications en desarrollo

Issues:

Question: Will paketo-buildpacks/java support the quarkus stacks ? · Issue #16

Issue #23559 · quarkusio/quarkus · GitHub

Ausencia de Implementación Reactiva en H2
Estado: Abierto, pendiente de Merge en el repositorio de Hibernate:

https://github.com/hibernate/hibernate-reactive/pull/1336

Issues:

Reactive H2 driver for Quarkus #20471
Add support for H2 · Issue #929 · hibernate/hibernate-reactive · GitHub

JsonView no ignora un campo no anotado:

Estado: Cerrado a falta de criterio de reproducibilidad del fallo.

Issue: Quarkus is not handling annontation @JsonView properly · Issue #7293

Solución temporal/Workaround:

Quarkus is not handling annontation @JsonView properly · Issue #7293

Quarkus CDI no tiene acceso a los Beans manejados dentro los callbacks de State de Pact

En los ejemplos de Pact, en los métodos anotados con @State no se tienen acceso a los Beans manejados como para preparar el estado para la respuesta que espera.

Issue: Quarkus CDI-managed beans not available in Pact state callbacks · Issue #22611

Solución temporal/Workaround:

Quarkus is not handling annontation @JsonView properly · Issue #7293

Panache Hibernate devuelve un error detached entity passed to persist que impide guardar la entidad

Estado: Solucionado

Causa: Valores definidos en el @Id en las entidades Comment y Post del ejemplo:

https://github.com/MasterCloudApps/2.2.Patrones-y-arquitecturas-de-servicios-web/tree/master/tema5/ejem1

Solución: Quitar los valores por defecto.

Extraída del Jira de Hibernate: Hibernate JIRA

Conclusiones: En Spring este problema no pasa.

Problemas de Integración con Mecanismos de Autorización y Autenticación

Quarkus no soporta un mecanismo de seguridad con anotaciones similar a Spring como @PreAuthorize de Spring y si lo hace es con una extensión que supone una capa de compatibilidad con Spring al igual que en JPA dispone una, pero no es una implementación propia y robusta como en el caso de Panache

Solución: Uso de Elytron JDBC junto con la anotación @RolesAllowed como manera segura de RBAC descrita en el Libro de Quarkus en el Capítulo 11.2

Descarte de: La solución de Elytron properties file descrita en el Libro de Quarkus en el Capítulo 11.1 por ser menos segura al almacenar las contraseñas en claro.

Integraciones: Elytron JDBC junto con con MicroProfile JWT Libro de Quarkus en el Capítulo 11.3 para el ejemplo de seguridad del tema 8 sobre JWT y Authenticathion Bearer.

Limitaciones descubiertas: Incapacidad de definir permisos sobre esos roles como hace Spring: @PreAuthorize("hasAuthority('book:write')")

Conscuencias de las limitaciones: Ausencia de Implementación role_example. Proporcionado ejemplo de basic_auth a cambio.

Descubrimientos: En caso de que se definan varios roles como una Lista en la entidad Usuario, a la hora de obtener los roles con elytron deberá hacerse un Inner Join ya que los roles estarán en otra tabla que si se define un solo string. Descubierto viendo la salida de las queries de Hibernate con la propiedad

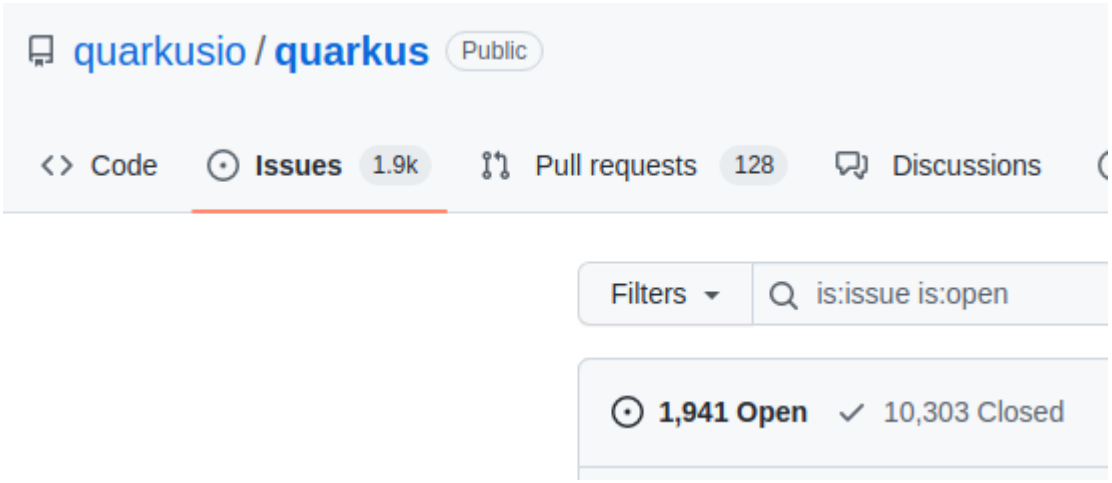
quarkus.security.jdbc.principal-query.sql=SELECT "password", "roles" FROM "user" INNER JOIN "User_roles" ON "id" = "User_id" WHERE "username"=?

Tests no encontrados, no Continous Testing ni Live Reload cuando se ejecuta la aplicación en modo Console y quarkus.test.continuous-testing=enabled

Issue: https://github.com/quarkusio/quarkus/issues/29790

Abierto por mi. Estado: Pendiente

Como podemos observar, no son pocos los problemas encontrados y ha habido que indagar mucho en distintos repositorios y páginas para analizar en detalle que esta pasando o si a alguien más le pasa. En los issues he observado ejemplos duplicados de problemas, quizás por regresiones y en otros casos no, y fallos concretos de una versión de la GraalVM. Podemos observar como en su repositorio aunque el número de problemas cerrados es alto, sigue teniendo un número muy elevado de issues abiertos, cerca del 20%. Si bien es cierto que para los menos de 4 años que lleva tiene muchos commits y está evolucionando muy rápido, al ser tanto código y ser esta sus circunstancias es más propenso a fallos, en NestJS no pasa aunque tiene muchos menos commits. Sí cabe destacar que la documentación parece bastante completa salvo en algún caso en concreto pero a veces uno se encuentra con fallos o limitaciones muy raros y no documentadas.



Capítulo 3: NestJS

3.1 Estado del Arte

NestJS es un framework escrito en TypeScript y altamente escalable diseñado para el desarrollo de aplicaciones web en el entorno de ejecución de Node.js. Fue creado el 25 de Octubre 2016 y desde entonces ha ganado popularidad en la comunidad de desarrollo de Node.js por su enfoque en la aplicación de patrones de diseño de software y su integración fluida con TypeScript.

NestJS se integra perfectamente con Node.js y sus módulos, lo que permite a los desarrolladores aprovechar la potencia de Node.js en sus aplicaciones de NestJS. Además, NestJS utiliza Express por debajo, lo que permite que los desarrolladores puedan acceder a todas las funcionalidades y herramientas de Express mientras trabajan con NestJS.

Una de las principales diferencias entre NestJS y Express es que NestJS está diseñado para ser utilizado con TypeScript en lugar de JavaScript puro. Esto permite a los desarrolladores hacer uso del tipado estático de TypeScript lo que puede mejorar la legibilidad y el mantenimiento del código. Además, NestJS viene con un conjunto de herramientas y librerías que facilitan la implementación de patrones de diseño en las aplicaciones, lo que lo diferencia de Express, que se enfoca más en ser una herramienta minimalista. Además, hay que tener en cuenta de que NestJS se integra con distintas soluciones de protocolos como Graphql, gRPC, webshocket, colas, etc. y proporciona mecanismos de seguridad de Autenticación y Autorización propios, mientras que Express requiere de librerías adicionales para integrarse con dichos protocolos.

Debido a su alta escalabilidad, a la naturaleza asíncrona de TypeScript podemos decir que es un framework ampliamente enfocado para usarse en arquitecturas de microservicios y por tanto en la nube aunque quizás no tanto como Quarkus.

3.2 Características

NestJS ofrece soporte nativo para REST, graphql y gRPC aunque también puede integrarse con librerías de terceros para trabajar con colas y mensajería, como RabbitMQ y Kafka.

Respecto a las bases de datos, NestJS viene incorporado para TypeORM, un ORM que le permite trabajar con bases de datos relacionales como MySQL y Postgresql aunque también puede integrarse con mongoose para trabajar con MongoDB y con sequelize para trabajar con otras bases de datos SQL.

NestJS emplea el servidor de GraphQL Apollo Server para ofrecer APIs bajo el protocolo de Graphql e incluye de manera similar a como lo hace Quakus un playground para probar estas APIs. También dispone de un cliente Graphql al igual que un cliente gRPC. Este playground se encontrará en la dirección /graphql dentro de la dirección de ejecución local que se esté empleando en el momento. Si bien es cierto que en el momento de escribir esta memoria apollo-server 3 va a ser deprecado NestJS aún no se ha integrado con Apollo Server 4 por lo que se podría seguir usando y tampoco hay otras alternativas.

Respecto a la organización del código, NestJS se organiza en módulos, una forma de organización en la que el código de una aplicación se proporciona en forma de unidades lógicas de funcionamiento. Los módulos en NestJS están inspirados en Angular y proporcionan una manera de organizar y agrupar funcionalidades en una aplicación NestJS.

Cada módulo en NestJS suele estar compuesto por uno o más controladores, servicios y otras dependencias. Los controladores son clases que manejan las solicitudes y envían las respuestas mientras que los servicios contienen lógica de negocio y pueden ser utilizados por varios controladores. Además, un módulo puede importar otros módulos y compartir sus dependencias con otros exportándolas ademá. Toda esta organización constituye la arquitectura habitual de una aplicación NestsJS, que como se puede observar no es muy diferente a una aplicación Java en Quarkus o Spring y que también se organiza también en clases.

Un ejemplo de un módulo se muestra a continuación:

```
@Module({
  imports: [DatabaseModule],
  controllers: [UserController],
  providers: [UserService],
  exports: [UserService],
})
```

export class UserModule {}

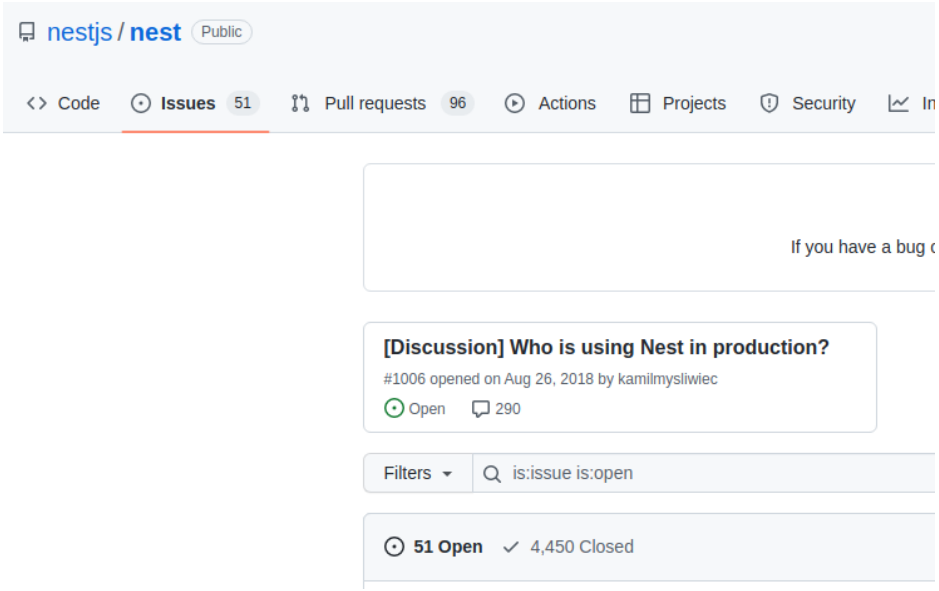
Para utilizar un módulo en una aplicación de NestJS, primero se debe exportar la clase del módulo y luego añadirlo a la lista de módulos al crear la aplicación.

En relación a los Controladores, en ellos se atienden las peticiones y se traslada la lógica de negocio al Service. Los controladores funcionan de manera similar a como lo hacen en Quakus y Spring y también permiten la Inyección de dependencias en ellos. La manera en la que NestJS lo hace es empleando la anotación @Injectable() en la clase susceptible de inyectar de dependencias e @Inject() e parámetro que debe ser inyectado. No obstante, cabe destacar que a diferencia de Quarkus, NestJS lo hace (o al menos los ejemplos que yo he visto) en el constructor de la clase y no como parámetro de la clase.

También emplea anotaciones como el verbo HTTP: @Post @Put @GET @DELETE @PATCH, la petición,@Req, el Body @Body en caso de recibir un objeto como parámetro, la respuesta @Res etc con los cual no lo hace muy diferente de otros frameworks.

Por último respecto a los test se realizan con jest y deben encontrarse a la misma altura de la fichero que testean y acabar como *.spec.ts donde el asterisco recibe el nombre de dicho fichero.

Cabe destacar que a diferencia de Quarkus no se han encontrado problemas, solo si acaso se puede conseguir, algún ejemplo de referencia difícil de encontrar o algún ejemplo incompleto pero nada más. Además a diferencia de Quarkus cuenta en este mismo momento con 51 issues abiertos de 4.450 lo cual es cerca de un 1%



Capítulo 4: Conclusiones y Trabajos Futuros

Como bien se ha visto, las diferencias de dichos frameworks entre sí no son tantas y una vez entendido los conceptos principales es más sencillo migrar de un framework a otro, o de uno de un lenguaje a otro de otro lenguaje. Además se ha visto como cada vez los frameworks son más propensos a dar primitivas más sencillas y a centrarse en resolver de otro modo problemas más complejos como la escalabilidad, la asincronía/reactividad, el arranque rápido, el bajo consumo de recurso, etc.

En este contexto podría plantearse como línea de trabajo futuro comparar el rendimiento de varios de estos frameworks en distintos entornos y circunstancias, local, on premise, en cloud, con arquitectura de microservicios, en monolito, en contenedores, etc. De esta forma podríamos centrarnos ya no solo en las primitivas que más que ser mejores en uno u otro framework son simplemente distintas y centrarse en los criterios medibles para decantarse por uno u otro framework.

También sería interesante comparar un ejemplo lo suficientemente complejo entre varios frameworks y una solución a bajo nivel que emplea distintas librerías pequeñas, pero en ningún momento un framework entero. Por ejemplo el caso de librerías como Express debería replantearse en este caso porque aunque es minimalista ofrece soluciones lo suficientemente buenas como para que resulte evidente que hay de código detrás que permite resolver más problemas de los que tu en verdad necesitas resolver y esta suele ser una crítica habitual a los frameworks.

En relación al contenido del máster también sería interesante ver algunos ejemplos en estos frameworks para ver que el alumno pueda observar de la misma forma que lo hace Spring con Java, como se integran con distintos servicios de la nube: RDS, s3, EC2 , etc. ya que, como se puede observar hasta el momento y sin ser un trabajo enfocado exclusivamente al despliegue en la nube, verdaderamente parece que están preparados para ello.

En resumen, con este trabajo y en base a los ejemplos hemos podido apreciar tanto en NestJS como en Quarkus, como ambos cumple con la definición de framework que mencionamos al principio.

Bibliografía

What is a Framework? Why We Use Software Frameworks - Code Institute Global

¿Qué es Quarkus?

Introducción a Quarkus: ¿qué es, características y por qué utilizarlo?

GraalVM Documentation

Contexts and Dependency Injection - Quarkus

Qute Templating Engine - Quarkus

Container Images - Quarkus

Building a Native Executable - Quarkus

Uber-JAR

Java y fluid interface

All configuration options - Quarkus

quarkusio/quarkus - Supersonic Subatomic Java.

NestJS

Scalable microservices in Node using Nest.JS | by Felipe Marques | Webera

GitHub - nestjs/nest: A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications on top of TypeScript & JavaScript (ES6, ES7, ES8) 🚀

Alex Soto Bueno & Jason Porter (Julio 2020). Quarkus Cookbook Kubernetes-Optimized Java Solutions. ISBN: 9781492062653

Anexos

Repositorio de Github

https://github.com/MasterCloudApps-Projects/alternative-frameworks